Inheritance relationships for disciplined software construction

Gardner, Tracy A.

Award date:
1999

Awarding institution:
University of Bath

INHERITANCE RELATIONSHIPS FOR DISCIPLINED SOFTWARE CONSTRUCTION

Submitted by Tracy A. Gardner

for the degree of

Doctor of Philosophy

of the University of Bath

1999

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

Inheritance Relationships for Disciplined Software Construction

Tracy A. Gardner

1999

Object–oriented inheritance has been in widespread use for a decade, it is now realised that although inheritance is a powerful modelling tool with many associated advantages, its benefits are not automatically conferred on systems that simply use it. In order to lead to readily understandable, easily maintainable systems with high levels of reuse, inheritance must be used well. This situation is further complicated by having multiple applications of inheritance that are considered to be valid modelling techniques. Although there are many guidelines pertaining to the use of inheritance, advice that is suitable for one form of inheritance may not be applicable to another. Inheritance, as currently available, is a difficult technique to master.

In this thesis, we develop a model of inheritance based around five fundamental inheritance relationships. Each relationship has a clear conceptual basis, representing a fundamental, specialised use of inheritance, the relationships — specialisation, variant, view, construction and evolution — can be used individually, or combined to achieve complex inheritance relationships. These five relationships are referred to as structured inheritance relationships (SIRs) since they introduce understandable and maintainable abstraction hierarchies rather than the 'spaghetti' inheritance hierarchies that result from using the same overloaded notion of inheritance to represent a number of underlying conceptual relationships.

The five SIRs are based on current uses of inheritance that have a sound conceptual basis. The underlying conceptual relationships are considered outside the restrictive context of current programming–language inheritance mechanisms and a detailed semantics is developed for each of the SIRs by considering the natural extension of these conceptual relationships. The resulting SIR model of inheritance replaces a confused notion of inheritance with five distinct conceptual relationships, supporting more precise modelling of systems and capturing the semantic intent of each use of inheritance within a system.

Acknowledgements

Without my valuable placement year (1993—1994) spent at Intera Information Technologies Ltd (now Enviros) it is unlikely that I would have embarked upon a PhD at all; it was during that year that I was first introduced to the object–oriented paradigm in depth. My colleagues there, in particular Dr Peter Robinson, Dr Mike Williams and Dr John Woods, provided a stimulating working environment in which the current way of doing things was never accepted by default and challenges to existing thinking were encouraged.

I would also like to acknowledge numerous fellow conference attendees at ECOOP '97 (especially the members of the PhDOOS workshop); OOPSLA '97 (especially the members the Doctoral Symposium); and Object Technology '97 and '98. The chance to discuss ideas with both researchers and practitioners at these events contributed markedly to the maturity of my research.

I would like to thank my supervisor Dr Claire Willis for her encouragement and motivation throughout my PhD. She was generous with her time and kept me on track. She also provided detailed comments on successive versions of my thesis — a time–consuming task for which I am extremely grateful.

I am greatly indebted to Dr Kevin Rutherford who has offered detailed criticism of my work, from both a theoretical and a practical perspective, on numerous occasions. Kevin has also provided much–appreciated constructive advice and encouragement regarding the process of completing a PhD.

Thanks to Dr Nic Doye for casting his mathematical eye over my work and providing valuable observations and suggestions. Over the years, Nic has always willingly answered my questions (some of them repeatedly, I'm sure). Cheers Nic.

A general thankyou goes out to my family, friends and colleagues who have always had confidence that I would get to the end of this, even when I wasn't so sure.

I'd also like to thank the developers of Linux, LaTeX, emacs and all of the other free software that has made working on my PhD a much more pleasurable experience that it might otherwise have been.

Thanks to Lucinda Williams and Jimmy Dale Gilmore for keeping me company and getting me through the final stages of writing up — even if you didn't know you were doing it.

Finally, I would like to thank my partner Beanz[1]. He has provided practical help, emotional support, and when all else failed, chocolate, and has helped me to keep things in perspective. Thanks Beanz, you are appreciated.

[1] Don't ask!

Contents

Chapter 1

Introduction

The designers of the Simula–67 language [Birtwhistle et al., 1973] introduced a powerful construct that has had an enormous impact on the process of software development for more than thirty years: the 'prefixing' construct of Simula–67 is the ancestor of the inheritance mechanisms found in modern object–oriented (OO) programming languages.

In the period since its inception inheritance has affected both the way in which programming is carried out and, significantly, the way in which software is modelled. Much of this change has been for the better since inheritance has been shown to be beneficial in reducing the conceptual overhead of understanding systems [p59–61 of Booch, 1994], for facilitating reuse [Meyer, 1987], and for supporting system maintenance and evolution [Liskov, 1987].

Although inheritance is a valuable technique when used well, an increasing recognition of the weaknesses of inheritance has developed in recent years. It is now widely accepted that simply using inheritance will not lead to well–designed, easily maintainable systems with high levels of reuse [Armstrong and Mitchell, 1994; Rumbaugh, 1993; Magnusson, 1991]. It is also understood that there is no single correct way of using inheritance; there are multiple applications of inheritance that are considered valid modelling techniques [Brachman, 1983; Halbert and O'Brien, 1987; Meyer, 1996; Edwards, 1993]. The power and flexibility of inheritance is both its strength and its weakness.

Some authors have even gone so far as to say that the problems with inheritance are so great that it should be avoided ([Chapter 7 of Szyperski, 1998] is subtitled 'how to avoid inheritance'), or that inheritance should be relegated to the realms of implementation and not used for the modelling of behaviour [p140 of Bennett, 1997]. In this thesis we show that the disadvantages are not an inherent characteristic of inheritance as a technique, rather they are the result of the lack of a structured way of using and applying inheritance.

Inheritance has allowed innovative software developers to explore the modelling of complex conceptual relationships between abstractions. The next step — the step addressed by this thesis — is to identify those relationships and move from an ill–defined, overloaded, intuitive understanding of inheritance to a number of well-defined relationships with clear conceptual foundations.

1.1 Objectives and Limitations

The goals of this thesis are:

1. The development of a comprehensive understanding of object–oriented inheritance including an overview of the current state–of–the–art in inheritance–related research.
2. The identification of the key conceptual relationships that are being modelled using inheritance and the design of a comprehensive framework for the effective development of software taking advantage of these relationships.
3. The demonstration of the power of the identified conceptual relationships both individually and in combination.

This thesis is not concerned with the development of a new object–oriented programming language. The focus is on the modelling stage of software development but with an awareness that the ultimate aim of such a model is to realize it.

To this end, a further objective is:

4. To show how systems designed using the developed framework can be implemented in object–oriented languages, and, if there are limitations to existing languages, to identify these and make suggestions as to how the situation could be improved.

Additionally, the focus is on conceptual issues in software modelling rather than the development of a mathematically–sound type system encompassing the various uses of inheritance. This is because such a mathematical model must be influenced by the conceptual relationships that we need to model, rather than the other way around. For this reason the development of such a type system is considered to be future work.

1.2 Approach

We are concerned with software development involving multiple systems within an application domain and large systems that will evolve over time. This focus reflects the environment in which much software is currently developed.

Experience with inheritance to date has shown that it is a powerful and flexible technique. However, these advantages come at a cost: inheritance is a difficult technique to use well and leads to a number of problems. These problems are a direct result of the lack of a structured framework in which to apply inheritance. In this thesis such a framework — which we will call the structured inheritance relationship (SIR) framework — is developed.

The framework is based on five structured inheritance relationships (SIRs) that have been developed to capture the applications of inheritance in software modelling.

The criteria for evaluating these relationships are that they must be

1. **necessary** — The relationships can be shown to have the characteristics of inheritance and to be necessary to model conceptual relationships that do occur in software systems.

2. **orthogonal** — Each relationship must serve a purpose that cannot be achieved using combinations of the remaining four relationships.

3. **sufficient** — The set of relationships covers the uses of inheritance that appear in existing classifications of inheritance, programming languages, design methodologies and technologies.

We will show that these criteria are met in practice through detailed case studies.

1.3 Conventions

This thesis makes use of terminology that is in common use in literature on object–oriented languages and methods. Unfortunately, the varied heritage of the object–oriented paradigm has led to a number of alternative terms being available for even the most fundamental concepts. It is therefore useful to introduce some basic terminology at this point. When discussing concepts, or categories of object, within a system the term **abstraction** will be used, an abstraction may describe a concept from the application domain or a concept introduced for implementation purposes. The terms **superclass** and **subclass** are used to refer to the participants in an inheritance relationship. The terms **type** and **implementation class** (or simply class) are used to refer to the interface and implementation aspects of a class, respectively. The procedures or functions associated with an implementation class are called **methods** and method interfaces are called **operations**. Where it is necessary to make a distinction, a programming language construct that implements inheritance is referred to as an **inheritance mechanism** while the conceptual relationship behind an application of an inheritance mechanism is referred to an **inheritance relationship**.

A glossary is provided to clarify the definitions of familiar terms (including those above) and as a point of reference for terms that may be less well–known. Less familiar terms are also explained when they are first introduced. This thesis also introduces new terminology as a part of the framework that is developed. This terminology is defined as it is introduced as also appears in a separate section of the glossary.

The diagrammatic notation for object–oriented designs is based on the unified modelling language (UML) [Fowler and Scott, 1997]. UML uses a named rectangle to represent a class. We specialise this notation so that a rectangle with a solid border denotes a type and a rectangle with a dashed border denotes an implementation class. The UML notation for generalisation (an open–ended arrow) is used to represent uses of inheritance with an underlying conceptual relationship. This notation is shown in Figure 1.1.

A Java–based pseudo code is used when discussing programming language issues. In general, access modifiers (public, private, etc), void return types, and method bodies are omitted for simplicity. When a general notion of inheritance is being discussed the pseudo keyword `inherits` is used rather than Java's `implements` or `extends`. A pseudo code example is given in Figure 1.2.

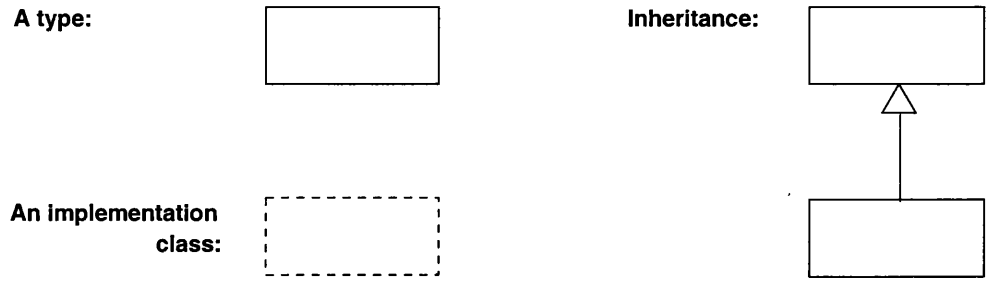

Figure 1.1: UML–based notation used throughout this thesis.

```
class Shape
{
   draw();
   move(int xincrement, int yincrement);
}

class Rectangle inherits Shape
{
   draw(); // override inherited draw method
}
```

Figure 1.2: An example of Java–based pseudo code.

1.4 Thesis Outline

Chapter 2: Current Understanding of Inheritance

In this chapter we consider the way in which inheritance has developed from the prefixing construct in Simula–67 to a modelling concept of great importance in the software development process.

This chapter provides the context for the rest of the thesis and explains the significance of the notion of inheritance, demonstrating that, although the problems associated with inheritance are considerable, it is preferable to solve these problems rather than managing without inheritance.

Chapter 3: Structured Inheritance Relationships (SIRs)

In this chapter we introduce the five SIRs that replace the general inheritance relationship that is currently in use.

We show, through examples, that the SIRs are necessary for modelling the conceptual relationships that are found in software domains (such as graphical applications, financial applications, etc). A clear conceptual description of each SIR relationship is provided.

We also show how and why the relationships have been modelled using inheritance mechanisms in mainstream object–oriented languages (such as Java, Eiffel and C++) and justify their classification as forms of inheritance. In each case we identify the characteristics of inheritance mechanisms that support the modelling of the relationship, we also highlight the areas in which current inheritance mechanisms are not ideally suited to modelling the SIRs.

Chapter 4: A New Model of Inheritance

Having established that the five SIRs are necessary, we develop the SIR model of inheritance to describe the SIRs in detail. We show how the SIRs, each supporting an orthogonal conceptual relationship, fit together to provide a complete model of inheritance.

The model developed in this chapter is based on the unified modelling language (UML) meta–model. We build a model by defining relationships between UML classes. This chapter also provides us with a graphical notation for representing SIR models based on the UML notation.

Chapter 5: Techniques for Structured Use of Inheritance

In this chapter we address the process by which SIR models should be developed. This includes a number of techniques (or patterns) that show how the SIRs can be employed in the modelling of system behaviour.

The techniques do not constitute a full software development methodology, they should be used in conjunction with an existing methodology such as one of [Jacobson et al., 1994; Rumbaugh et al., 1991; Booch, 1994]. The SIR techniques should be employed when potential inheritance relationships are identified using an existing methodology.

Chapter 6: Case Studies: Applying the SIR Framework

In this chapter we put the SIR relationships introduced in Chapter 3 and defined in Chapter 4, and the techniques described in Chapter 5 to the test by designing a software system. We show that this approach allows us to model the system more precisely, leading to a more structured design. The description of the system in terms of the SIRs is also more straightforward than a corresponding description using a single general form of inheritance which would also require the use of other techniques due to the comparatively weaker modelling power of the current notion of inheritance.

Additionally, we apply the SIR framework to two well–known modelling problems that were introduced in Chapter 2: the square/rectangle problem and the point/coloured–point problem. We show that these problems can be explained conceptually in terms of the SIR model. The analysis provides the basis for alternative modelling solutions to the problems for use in different contexts.

Chapter 7: Implementation Techniques

The aim of this chapter is to show that the SIRs can be implemented within a class–based and, (for the most part) statically–typed language. Since the five SIRs are extensions of the conceptual relationships that have been modelled using the inheritance mechanisms of existing programming languages, we can implement certain aspects of the SIR model using the inheritance mechanisms available in programming languages such as C++, Eiffel and Java. In this chapter we consider the current level of support for each of the SIRs in the Java language.

Since the SIRs are not restricted by the semantics of inheritance in Java, it is not possible to adequately support the SIR model within the current Java language. Therefore, we also consider how Java could be extended to provide full support for the SIRs, a number of Java language extensions and meta–object protocol techniques are employed to outline possible implementation approaches.

Programming language constructs are not the only way of supporting the SIR framework, we also consider the rôle of CASE tools in providing a suitable environment for SIR modelling.

Chapter 8: Conclusion

In the final chapter the model of inheritance developed in this thesis is reviewed, the SIR model is contrasted with related work, and directions for future work are discussed.

We conclude that the problems currently associated with inheritance are not inevitable consequences of its application. The usefulness of inheritance can be greatly enhanced by applying it within the structured framework provided by the SIR framework.

Chapter 2

Current Understanding of Inheritance

Although inheritance mechanisms have been available for over thirty years, and in mainstream use for a decade, there are still many unresolved issues surrounding the use of inheritance. Object–oriented inheritance has been influenced by ideas from a number of research disciplines including artificial intelligence, biological classification and database theory, this has lead to a rich but complicated set of ideas which are collectively referred to as 'inheritance'. This chapter considers the evolution of inheritance in software development, from its origins in the Simula–67 language through to the current understanding of the relationships, principles and mechanisms known as inheritance.

2.1 The Origin of Inheritance

A number of disciplines have contributed to the current concept and constructs known as inheritance. These include cognitive science, philosophy and artificial intelligence as well as abstract data type theory, data modelling and programming language development.

The classification hierarchy aspect of inheritance has a long history that can be traced back to the Linnaen classification of the natural world and beyond. The notion of more specific classes (or categories) inheriting properties from more general ones is the concept upon which object–oriented inheritance is based.

Inheritance as a modelling technique has been employed in the study of artificial intelligence (AI). Both semantic networks [Quillian, 1967] and frames [Minsky, 1981] use the approach of defining more specialised entities by inheriting properties from more general entities. Inheritance in AI systems usually occurs at the object or instance level rather than at the class level as in object–oriented programming.

The idea of creating sub–entities that specialise existing entities is well established in relational data modelling. However, as with the use of inheritance in artificial intelligence, such modelling is restricted to the inheritance of data and the inheritance of behaviour is not supported.

The key concepts of the object–oriented paradigm, including inheritance, first appeared in the programming language Simula [Birtwhistle et al., 1973] in 1967. This version of Simula is often referred to as Simula–67 to distinguish it from later versions of the language.

Simula, as its name suggests, was created as a discrete event simulation language. In this domain, more so than in other domains for which software had been developed, the elements of the program correspond closely to the real–world objects that are being modelled. In such a domain it is natural to describe a system in terms of its constituent objects rather than in terms of its overall behaviour. The developers realised that this approach could be applied to the development of software systems in general.

Simula introduced the concept of packages with instances (which we would now call classes) and also a prefixing construct which provided single inheritance. The conceptual rôle which the prefixing construct was intended to play was clear early on [Birtwhistle et al., 1973]:

> Each level inherits the concepts defined in previous levels and concentrates on extending them a stage further. The packages resulting at any level may be used in their own right, or employed as foundations for new levels. This means that at each stage we can concentrate upon a few related concepts and need not clutter up our thinking apparatus with other details.

After Simula–67, other object–oriented languages were developed with Smalltalk [Goldberg and Robson, 1983] being the most prominent early example. Smalltalk was developed as an object–oriented language from the start. The key difference between Simula and Smalltalk is that whereas Simula is statically–typed, Smalltalk is dynamically–typed. In dynamically–typed languages inheritance is not required for substitutability so the emphasis of inheritance is on reuse. As with Simula, Smalltalk supports only single inheritance.

The early 1980s is notable for the introduction of two statically typed, object–oriented languages: C++ [Stroustrup, 1991] and Eiffel [Meyer, 1997]. Eiffel was developed as an object–oriented language whereas C++ was based on the existing C language. C++ is the language that brought the object–oriented paradigm to industry on a large scale, and is still the most widely used object–oriented language today.

The inheritance mechanisms of C++ and Eiffel have much in common. Both languages combine interface and implementation into a class construct with inheritance leading to both code reuse and polymorphism. In both cases, support is provided for method overriding and type–safe method redefinition. Multiple inheritance is available in both languages and both provide a notion of abstract classes that cannot be instantiated directly but must be subclassed to provide concrete classes. Although the features of inheritance in Eiffel and C++ are broadly similar they differ significantly in their detailed semantics, as we will see throughout this chapter.

As object–oriented programming languages were developing, so was the theory of abstract data types (ADTs) which has also influenced the current understanding of inheritance. ADT theory began in the 1970s with the work of Parnas, Liskov, Zilles and others [Parnas, 1972; Liskov, 1974; Liskov and

Zilles, 1975]. Simula, and thus the start of the object–oriented paradigm, therefore predates the first work on abstract data types. However, the object–oriented paradigm did not become mainstream until the late 1980s/early 1990s, and in the intervening time period abstract data type theory had matured to the point where it was able to influence object–orientation.

An abstract data type (ADT) specification defines a set of objects in terms of the operations that can be carried out on them; an ADT specification does not include implementation details. ADTs are specified in terms of the operations that can be invoked on objects by clients. Here abstract means that the representation is not specified so the same ADT can be represented in different ways. For example, we could have a Triple ADT specified in terms of operations on its first, second and third slots; this ADT could be represented by three separate data values, or by an array with three elements.

Early work on the theory of ADTs did not include a notion of inheritance. However, ADT theory provided the foundation for formally understanding the notion of behavioural subtypes. The behavioural subtyping principle has come to be known as the Liskov Substitution Principle (LSP) and is well–known to object–oriented developers. The principle was first stated, informally, in [Liskov, 1987] as:

> What is wanted is something like the following substitution property: If for each object o_1, of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .

This property is often weakened in common usage to include cases where a client expecting program P without the substitution would be satisfied with P after the substitution. This principle was developed in [Liskov and Wing, 1994] to give a definition of behavioural substitutability:

> *Subtype Requirement*: Let $\phi(x)$ be a provable property about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

This (Liskov and Wing) subtype requirement is stronger than the subtype relationship enforced by programming languages such as C++ and Java which is purely syntactic. The Liskov and Wing subtype requirement ensures that subtypes are *behaviourally compatible* with supertypes. Eiffel provides direct support for the LSP: the interface between a client and server is seen as a contract which can be specified within the Eiffel language, a subclass inherits the contract of its superclass and must honour it (this is discussed in more detail in Section 2.5).

The late 1980s and the early 1990s saw the rise of the object–oriented design method [Booch, 1994; Rumbaugh et al., 1991; Jacobson et al., 1994] and with it the popularisation of object–oriented technology. Unlike earlier forms of inheritance in AI and database modelling, object–oriented methods support the modelling of the inheritance of behaviour as well as data.

In OO methods, the 'is–a' rule is widely used as a litmus test for inheritance. That is: a class Y should inherit from a class X if, and only if, Y *is–a* X. According to this rule, we can say 'an

Apple Tree is–a Tree', so inheritance is appropriate, we cannot say that 'an Apple Pie is an Apple', so inheritance should not be used in this case. The simplicity of this rule has led to its widespread adoption.

The LSP as a basis for subtyping inheritance is also promoted, in some form, by all of the object–oriented methods. This rule can also be stated simply as: wherever a supertype instance is required a subtype instance may be used. Under this approach, the test would be 'if a client asks for a Tree and gets an Apple Tree, will they be satisfied?', the answer is yes, so inheritance should be used. On the other hand, 'if a client asks for an Apple and gets an Apple Pie, will they be satisfied?' results in a negative answer so inheritance should not be used.

Inheritance without a conceptual 'is–a' relationship is often advised against. The Object–Oriented Software Construction (OOSC) methodology (as described in [Meyer, 1997]) is an exception to this approach, inheritance relationships that do not lead to conceptual is–a relationships are also considered valuable, provided they meet the design–by–contract principles (discussed in Section 2.5) which are similar to the LSP. For example, a graphical window application might inherit from a tree data–structure in order to handle its hierarchy of child windows. We cannot say that a window is a tree but the window class maintains the contract of the tree class so the use of inheritance would be considered appropriate.

Inheritance without subtyping is generally considered to be undesirable although C++, the most widely used object–oriented language, supports such inheritance, termed 'private inheritance'. Private inheritance can be used in cases where inheritance should not lead to a subtyping relationship, as may be the case with the window/tree example above: although a window has all the behaviour associated with a tree, there may be no requirement to use window instances polymorphically with other tree instances. Although this example may be seen as a legitimate use of inheritance by some methodologists, private inheritance also supports the development of structures where the inherited operations do not properly belong to the subclass. For example, a word processor class inheriting from string because it wants to be able to use string manipulation functions. Although convenient, such relationships make systems difficult to maintain and understand.

Multiple inheritance, in which a subclass has two or more superclasses, is treated with caution in most object–oriented methodologies. This is typically for practical rather than conceptual reasons. Multiple inheritance leads to the name clash problem where two properties with the same name are inherited into a subclass. Although object–oriented languages supporting multiple inheritance have conflict resolution mechanisms to handle such situations, it is often presented as a problem. Another reason for avoiding multiple inheritance is complexity, systems that use multiple inheritance are considered more difficult to understand than those that use only single inheritance. Few methodologists argue against multiple inheritance altogether though. This view is summed up by Booch [Booch, 1994]: 'we find multiple inheritance to be like a parachute: you don't always need it but, when you do, you're really happy to have it on hand.' Again, the OOSC approach differs here, certain kinds of multiple inheritance are considered valuable modelling techniques.

2.2 What is Inheritance?

Software developers tend to have an intuitive understanding of the concept of inheritance, however they do not necessarily have the same intuitive understanding. This is noted in [Sakkinen, 1989]:

> 'Inheritance' is an appealing word because its meaning in object–oriented programming (OOP) is so analogous to its usual meaning, which in turn is familiar to everybody. Still, probably because of this intuitiveness, there seems to be no common definition of 'inheritance'.

Although this observation was made ten years ago, it still holds today.

In this section we consider a number of proposed definitions of (software) inheritance, the definitions vary as to which aspect of inheritance they emphasise. As we will see in Section 2.4, where problems with inheritance are discussed, these different interpretations of inheritance are not always compatible.

We start with the description associated with the Simula–67 language.

> Each level inherits the concepts defined in previous levels and concentrates on extending them a stage further. The packages resulting at any level may be used in their own right, or employed as foundations for new levels. This means that at each stage we can concentrate upon a few related concepts and need not clutter up our thinking apparatus with other details.

This is still recognisable as a description of inheritance (although Simula–67 used the term prefixing rather than inheritance). This definition focuses on the abstraction aspect of inheritance which allows a subclass to concentrate on the properties that distinguish it from other instances of the superclass.

Next we consider the definitions of inheritance proposed by Booch, Rumbaugh and Jacobson, key developers and proponents of three widely used object–oriented software design methods.

Jacobson focuses on inheritance as a mechanism for sharing:

> If class B **inherits** class A, then both the operations and the information structure described in class A will become part of class B. [Jacobson et al., 1994]

Booch is concerned with inheritance as a conceptual relationship between classes:

> **inheritance** A relationship among classes, wherein one class shares the structure or behaviour defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines an "is–a" hierarchy among classes in which a subclass inherits from one or more generalized superclasses; a subclass typically specializes its superclasses by augmenting or redefining existing structure and behaviour. [Booch, 1994]

Rumbaugh's definition appears similar:

> *Generalization* is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is called a *subclass* ... Each subclass is said to inherit the features of its superclass. ... Generalization is sometimes called the "is–a" relationship because each instance of a subclass is an instance of the superclass as well. [Rumbaugh et al., 1991]

But, note that Booch uses is–a at the class level, whereas Rumbaugh discusses the is–a relationship in terms of instances. This is a common confusion associated with inheritance. When we say Dog inherits from Mammal, are we saying that the Dog concept is a kind of Mammal concept or that every Dog is a Mammal? The first would suggest that wherever the Mammal type appears in a program the Dog type could be used whereas the second would indicate instance–level substitutability. We return to this issue in Section 2.4 when we consider the different kinds of substitutability that can result from inheritance relationships.

Although the definitions presented so far differ, they all agree that inheritance is to do with the characteristics of a superclass becoming characteristics of a subclass. The issue of whether subclass instances should be substitutable for superclass instances is less clear.

The unified modelling language (UML) is a unification of the modelling languages and concepts that constitute the methods of Rumbaugh, Booch and Jacobson (and others). UML offers the following definition of inheritance (from the Glossary of the UML Semantics document [Booch et al., 1997]):

> The mechanism by which more specific elements incorporate structure and behaviour of more general elements related by behaviour.

That is, inheritance does not imply substitutability, simply behavioural similarity. The term generalization is used in UML to describe the is–a relationship: 'A taxonomic relationship between a more general and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed.' This is similar to the Liskov and Wing subtype requirement in that it implies instance level substitutability.

Although the UML definition of inheritance is general enough to accommodate the inheritance of characteristics without a subtyping relationship, the definition of generalisation seems to prevent this. However, UML does allow for private generalization which does not have this property.

What we do gain from the UML definition of generalization however, is that the entities involved in an inheritance relationship need not be classes. Although here we will focus on inheritance relationships between classes for simplicity, it is important to note that inheritance can also apply to other units. Much of the work in this thesis will therefore be relevant beyond the level of individual classes.

Another angle on inheritance is provided by the following definition of inheritance which was developed at the first Foundations of Object–Oriented Languages workshop in 1993 [Black and Palsberg, 1994]: 'a mechanism for making one class or type from another, in which self is late–bound'.

This definition focuses on the idea that inherited operations execute within the subclass rather than within the superclass. This means that the subclass version of an overridden method is always executed in preference to a superclass version, even when executed via an inherited superclass method.

The differences between the definitions of inheritance presented here are indicative of the complexity and immaturity of the notion of inheritance. In Chapter 3, we develop a definition of inheritance to be used within the SIR model and throughout the remainder of this thesis.

2.3 Advantages of Inheritance

When object–oriented languages began to break through into mainstream software development, inheritance was promoted as a technique that would revolutionize software development. Though the vision of inheritance as a magic bullet has not been realized, the advantages of inheritance are still compelling. We now introduce the key advantages of object–oriented inheritance.

Reduction in Complexity The reduction in complexity that can be achieved through the use of inheritance is a key advantage. Rumbaugh et al [Rumbaugh et al., 1991] go so far as to say that 'the most important use of inheritance ... is the conceptual simplicity that comes from reducing the number of independent features in a system'.

Inheritance reduces the conceptual overhead in understanding an abstraction within a system. Cox [Cox, 1986] sums up this notion as follows: 'inheritance makes it possible to define new software in the same way we would introduce any concept to a newcomer, by comparing it with something that is already familiar.' A related advantage is noted by Booch [Booch, 1994]: 'inheritance lets us state our abstractions with an economy of expression. Indeed neglecting the "is–a" hierarchies that exist can lead to bloated, inelegant designs.'

Conceptual simplicity is a highly desirable quality in software development, especially as systems are continually increasing in size and complexity.

Inclusion Polymorphism In most mainstream object–oriented languages (C++, Java [Arnold and Gosling, 1996], Eiffel) a type and its implementation is combined into a single unit, the class, this means that inheritance leads to a subtyping relationship in which instances of a subtype can be used as if they were instances of a supertype.

This relationship makes inclusion polymorphism possible. Instances of all subclasses of a class (including the class itself) can be accessed polymorphically by a client. That is, clients need only know about the existence of the superclass in order to be able to deal with subclass instances as well.

Each subclass can specialise methods and the most specific method for a particular instance will be invoked at run–time. For example, we might have a window object which contains a number of shapes. The shapes may actually be instances of rectangle, circle and triangle subclasses of the shape

abstraction. The window abstraction need only know about the general shape abstraction to invoke the draw operation on each of its contained shape objects. When the draw operation is invoked on an object the draw method appropriate to that object will be invoked, this will be different for squares, circles and triangles. Since the window abstraction does not need to know about the specific subclasses of shape it will be able to deal with new shape subclasses, such as five–pointed star and heart, as they are introduced.

The type of the variable in which an object is held, which may be a supertype variable, is said to be the **static type** of the object within that context. The actual, or most specific, type of the object is referred to as its **dynamic type**. The correct method to invoke, in the case of inclusion polymorphism, is the method associated with the dynamic type of the object.

Increased Consistency Inheritance also encourages consistency across similar abstractions as described by Cox [Cox, 1986]:

> Without inheritance, every class would be a free–standing unit, each developed from the ground up. Different classes would bear no relationship with one another, since the developer of each provides methods in whatever manner he chooses. Any consistency across classes is the result of discipline on the part of the programmers.

Note that although such consistency is required for polymorphism it is also advantageous across abstractions that are not intended to be used polymorphically.

The rôle of inheritance in providing consistency for software developers is recognised: the effort to understand a new abstraction is reduced if the developer has already used similar abstractions. Inheritance also provides consistency for end–users: it can be used to ensure that aspects of a system that are conceptually similar to the user behave in the same way. It is often the case that support for similar functionality is inconsistent, even within a particular system. This is confusing to the end–user and should not happen in a well–designed system. The use of inheritance to relate concepts can help to avoid such problems since the behaviours shared by conceptually similar abstractions can be implemented in a common superclass.

Superclass Reuse Reuse has been widely promoted as an advantage of inheritance. The most obvious form of reuse supported by inheritance is the reuse of superclass code. Jacobson [Jacobson et al., 1994] describes this form of reuse:

> The most common reason for using inheritance is that it simplifies the reuse of code. Reuse can, in principle, occur in two different ways in combination with inheritance. The first is that two classes are found to have similar parts; these parts are extracted and placed in an abstract class, which both the original classes inherit. This abstract class represents the common parts of both classes and need not always be meaningful in itself. The other way to reuse is to start from a class library. Find a class which has the operations you need, inherit this class and make the required modifications.

It is not just the code itself that is valuable but the fact that effort has been put into developing that code. As well as reusing code, inheritance supports the reuse of the development effort that led to that code.

It is not simply reuse of superclass code that inheritance gives us, it is the ability to inherit that code into a new context. Inherited methods may be overridden to provide specialised behaviour, and crucially, other inherited methods will use the specialised version of the method. This is what distinguishes reuse through inheritance from reuse by message forwarding (or composition).

Client Reuse A less well understood but potentially more important form of reuse gained from inheritance is the reuse of client code. When inheritance is used to create a subtype, all of the existing code that handles supertype instances can be reused to handle subtype instances. This form of inheritance–related reuse is discussed in [Biddle and Tempero, 1995]. Client reuse is significant since the amount of client code reused when introducing a subclass is potentially much greater than than the superclass code which is inherited by the subclass.

Wider Reuse As well as the reuse of code, the use of inheritance has far wider implications for reuse. When reusing a superclass the subclass also benefits from the analysis and design that went into the superclass.

Reused code has already been tested so only new code and modifications to existing code need to be tested. Additionally, code that has already been used is likely to be more stable than newly developed code.

As already noted, inheritance leads to consistency, this allows developers and end–users comprehension of an abstraction to be reused when it is extended via inheritance.

Propagation of Modifications An important 'side–effect' of using inheritance to relate abstractions is that modifications (improvements, extensions or error corrections) that are made at a particular level in the hierarchy are automatically propagated to all descendents.

For example, a billing application for a utility company might have a notion of a Consumer Account with subclasses for standard accounts, premiums accounts, low usage accounts, and so on. Suppose that such accounts currently only allow a single account holder but that there is a requirement to allow multiple account holders. The Consumer Account superclass can be modified and the subclasses will all inherit the new functionality and meet the new requirement automatically.

2.4 Problems with Inheritance

The benefits of inheritance — such as, superclass and client reuse, inclusion polymorphism, and reduction in complexity — have led to inheritance being widely used. Inheritance is clearly powerful, but in practice its benefits are not always evident. The complexities and subtleties of the concept of inheritance have also resulted in a number of problems that prevent inheritance achieving its perceived potential. We now consider the best known of the problems associated with inheritance.

Fragile Base Class Problem A well–known problem with using inheritance is that the advantages of the propagation of changes to subclasses come at a cost. Some changes to a superclass (also known as a base class) may 'break' subclasses. Since the superclass does not know about its subclasses this cannot be predicted.

A class has an obligation to its clients to keep its public interface consistent since changes to its public interface will obviously affect clients. The obligation to maintain internal interfaces is less clear. The primary obligation of a superclass is to its direct clients, rather than its subclasses. Modifications to meet changes in client requirements may have negative repercussions for subclasses.

The Yo–Yo Problem The yo–yo problem is a colourful name for the difficulty in understanding the behaviour of classes in deep inheritance hierarchies. A method inherited from a class several layers up in the inheritance hierarchy may invoke a method that is overridden in the subclass but also invokes the method in the superclass two layers up. In other words, the call–tree for a method can get extremely complicated. This complexity must be weighed against the conceptual simplicity that inheritance is supposed to offer.

Subclassing $\neq$ Subtyping $\neq$ Is–a The problem with an intuitive understanding of inheritance is that not everyone has the same intuitive understanding, and even worse, some interpretations of inheritance are incompatible with others.

In [LaLonde and Pugh, 1991] three distinct interpretations of inheritance are discussed: subclassing, subtyping and 'is–a'. Subclassing refers to the inheritance of implementation; subtyping refers to the relationship required to achieve instance–level object substitutability; and is–a is the conceptual specialisation of a superclass. Although the interpretations of inheritance often coexist they can lead to different inheritance hierarchies under certain circumstances. If only one form of inheritance is provided in a language then it must either select one of these interpretations of inheritance or support some form of compromise. Current inheritance mechanisms combine both subclassing and subtyping, although modelling is typically carried out using the is–a relationship. Several of the inheritance–related problems below result from this confusion in the meaning of inheritance.

Restriction — Square/Rectangle Problem The square/rectangle problem is probably the best known of a class of inheritance modelling problems that prove problematic for beginners and more experienced users. It is intuitive to create a Square class as a subclass of a Rectangle class since a square is–a rectangle. The problem is that a Rectangle class may have operations such as stretch–x and stretch–y that are inherited by the Square subclass but which do not apply to square instances (see Figure 2.1).

The reason that the problem occurs is well–understood. The statement 'a square is a rectangle' means that any square instance is also a valid rectangle instance; that is, the set of all squares is a subset of the set of all rectangles. When we apply rectangle operations to a rectangle the result is a rectangle. When we apply rectangle operations to a square the result will be a rectangle, but it will not necessarily be a square. This is a problem when dealing with mutable classes, that is

```
class Rectangle
{
   // constraint: x > 0 and y > 0

   private int x;
   private int y;

   stretchX(int xMultiplier);
   stretchY(int yMultiplier);
}

class Square inherits Rectangle
{
    // constraint: x==y
}
```

Figure 2.1: Square as a subclass of Rectangle.

classes with instances that can change their values over time. If the classes were immutable, then a stretch operation could return a new rectangle object, even if applied to a square. When the stretch operation is applied to a mutable object, the existing object must be modified as a result of the stretch. If this object is a square and the result needs to be a rectangle then we have a problem: the square class cannot accommodate a rectangle value. Object–oriented classes are typically mutable so this is a serious problem.

Object–oriented texts often introduce inheritance as a subset/superset relationship. It is easy to misunderstand this to mean only that the set of subclass instances is a subset of the set of superclass instances. Such a relationship is not sufficient for substitutability. The LSP is a relationship on objects, not on sets of objects. Substitutability requires that the set of behaviours on a subclass instance is a superset of the set of behaviours on the superclass. This is not always compatible with restriction — introducing further constraints in a subclass — which is one of the main ways of specifying the difference between a subclass and its superclass.

Extension — Point/Coloured–Point Problem Inheritance often involves adding new properties to an abstraction to create a new abstraction that is everything the superclass abstraction was, and more.

Consider a 2–D point abstraction with an equality operation such that two points are equal if they have the same values for their x and y coordinates. We can subclass this abstraction to create a coloured–point abstraction that as well as having a location in 2–D space, also has a colour (Figure 2.2). All appears fine until we consider the behaviour of the equality operation on coloured points. Within the subclass context, it makes sense for this operation to test for colour in addition to location. This is only possible if the argument to the equality method is another coloured–point. But, the subclass cannot narrow the type of the argument to the equality method to accept only coloured points since superclass clients will expect to be able to pass a 2-D point as an argument to the equality method, even if they are actually dealing with a coloured–point.

In C++, it is possible for the subclass to define an equality method with a specialised argument type. However, this overloads rather than replaces the superclass equality method. The subclass must also support the inherited operation which accepts points, so the specialised subclass method

```
class Point
{
   private int x;
   private int y;

   move(int newx, int newy);

   equals(Point p);
}

class ColouredPoint inherits Point
{
   private Colour col;

   changeColour(Colour newcol);
}
```

Figure 2.2: Coloured point inherits an equality method which accepts points.

coexists with (or overloads) the inherited superclass method. The inheritance mechanism of C++ gives us no way of saying that the argument to a coloured–point equality method must be another coloured–point. The same is true of Java.

This problem occurs because the relationship we would like to model is that the set of all coloured–points is substitutable for the set of all points, in other words, we want class–level substitutability rather than instance level substitutability. But, inheritance only gives us instance–level substitutability. In Section 2.5 we discuss techniques to support the implementation of such relationships.

Multiple Receivers The receiver of an operation invocation is the object to which the operation is applied. The correct method to invoke is determined by the dynamic type of the receiver. This means that, in the presence of subtyping inheritance, the subclass version of a method will be invoked if it exists. In some cases, the correct method also needs to depend on the dynamic type of one or more of the arguments to the method, in other words there are multiple receivers. This is only relevant in the presence of subtyping: without subtyping the static and dynamic types of an object will always be the same, the static type and dynamic type of an object can only differ when a subtype object is held in a supertype variable.

Consider an application for setting up connections between modems. There are several brands of modem and some brands of modem support their own proprietary communication protocol as well as a generic one. A connection between two modems of the same brand must be conducted using the proprietary protocol rather than the generic one. At first this seems similar to the extension problem above, but in this case we do not want a connection between two modems of different types (brands) to fail type checking (as we did with mixed comparisons of points and coloured–points), we want a different connection protocol to be initiated.

The dynamic type of each of the arguments to the connect method must be known in order to determine the correct method to be invoked. Most object–oriented languages, including C++, Java and Eiffel, only allow the dynamic type of the receiver to contribute to method selection, this is known as single dispatch. In single dispatch, the static (declared) type of arguments other than the receiver is used in method selection. This means that even if specialised connect methods are

defined on subclasses for appropriate combinations of modems, these methods will not be invoked when the arguments are held in supertype variables (see Figure 2.3). The generic method defined on the superclass will be used, regardless of the dynamic types of the arguments.

```
class Modem
{
   connect(Modem m); // generic connection method
}

class XModem inherits Modem
{
   connect(XModem x); // XModem-specific connection method
}

Modem x1 = new XModem();
Modem x2 = new XModem();

x1.connect(x2);      // invokes generic connection method
```

Figure 2.3: Single dispatching is not sufficient for the Modem problem.

This problem, and the more general problem of being able to specify precisely the method to be invoked for any combination of argument types (subject to subtyping rules) cannot be handled directly by mainstream object–oriented languages. Multiple dispatch is required to support such behaviour, this is discussed in Section 2.5.

Exception to the Rule Inheritance is supposed to allow abstractions to be explained in terms of other similar abstractions but there is a caveat that is not normally present when explaining one abstraction in terms of another: the subclass abstraction can only add information, it cannot take it away. This means that inheritance often cannot be used in an intuitive way. Consider a Bird superclass which defines characteristics of birds (two-legs, beak, can fly) and then consider inheriting from Bird to create a Penguin subclass. Unfortunately, penguins are an exception to the rule that all birds can fly. Since the superclass states that Birds can fly, Penguin cannot be a subclass of Bird although intuition tells us it should be.

This is a consequence of the way inheritance is used in object–oriented languages: it is monotonic. That is, information obtained from a superclass applies to all direct instances and subclass instances. Semantic networks, on the other hand, often support non–monotonic reasoning: statements about parents are taken to be defaults to be used in the absence of contradictory evidence. In such a system it is only safe to assume that a particular bird can fly if there is no evidence to the contrary.

Non–monotonic reasoning is contrary to the goals of type–safety and is not generally supported by object–oriented inheritance. The main exception is in Eiffel which allows unsuitable methods to be disinherited; in order to maintain type safety, complex type checking is required to ensure that disinherited methods are not called on instances that do not support them.

Spaghetti Inheritance One of the key advantages of inheritance should be the reduction in complexity that arises from the hierarchical organisation of abstractions. Unfortunately this advantage can be lost when inheritance is used heavily for code reuse (another key advantage of inheritance). If

inheritance is used for code reuse then the inheritance hierarchy is a mixture of conceptual and non–conceptual relationships. Instead of being simpler to understand the resulting 'spaghetti inheritance' structure is potentially more difficult to understand that a set of stand–alone abstractions.

There have been a number of suggestions as to how to solve this problem with perhaps the most popular being that inheritance should only be used where there is a conceptual 'is–a' relationship between abstractions [Magnusson, 1991]. Another suggestion is the separation of the subtyping hierarchy from the subclassing hierarchy [Porter, 1992]. This approach leads to a conceptual hierarchy in which abstractions are related by subtyping, and a separate hierarchy in which code reuse occurs.

The latter approach requires the acknowledgement that a single relationship is not sufficient to capture all the benefits of inheritance.

Inheritance is Difficult to Use Well Inheritance brings benefits such as reduction in complexity, inclusion polymorphism and reuse. But, it is also recognised that to obtain these benefits software developers must use inheritance well. In [Firesmith, 1995], the author notes that 'High–quality inheritance structures are required if object technology is to achieve its promise of greatly increased extensibility, maintainability, reusability and understandability. It is not enough to merely create numerous subclasses from existing superclasses; one should also build good overall architectures exhibiting the best uses of inheritance.'.

Unfortunately, the meaning of 'best uses of inheritance' is not immediately obvious. A number of authors, including Firesmith [Firesmith, 1995] and Kuo [Kuo, 1995] have developed guidelines for using inheritance well. Both Seidewitz [Seidewitz, 1996] and Sakkinen [Sakkinen, 1989] make it clear that inheritance offers too much freedom to software developers.

Inheritance, in the form currently available in mainstream programming languages, is obviously a difficult technique to master and apply effectively.

Multiple Kinds of Inheritance One of the most commonly cited causes of the difficulty involved in making good use of inheritance is that there are several different kinds of inheritance. Inheritance can be used to model a number of different conceptual relationships as discussed in [Brachman, 1983], [Meyer, 1996] and [Edwards, 1993].

While this is not a problem in itself it does currently lead to problems. Firstly, inheritance mechanisms cannot precisely model multiple conceptual relationships, each with different semantics. This means that current inheritance mechanisms are a compromise. Secondly, even if the developer is aware of the form of inheritance that is being used when a class hierarchy is developed, this information is lost when it is mapped to an all–purpose inheritance mechanism.

2.5 Understanding Inheritance

In this section the semantics of inheritance is considered in detail. Currently, the precise semantics of inheritance is controlled by programming language mechanisms. Although inheritance can be

seen as a conceptual relationship, in practice its design–level semantics is strongly influenced by the programming language that will be used to implement the design.

A number of issues that influence the semantics of inheritance are discussed in the following sections. This provides an understanding of the way in which inheritance operates in practice, and the ways in which mainstream inheritance mechanisms can currently be used.

Static versus Dynamic Typing In a statically–typed language it is possible for the compiler to guarantee that a program will never get into a situation where an operation is invoked on an object that has no corresponding method. Dynamically–typed languages, on the other hand, make no such guarantee. For example, a Smalltalk operation invocation, known as a message send, can result in 'Message not understood' if the object has no method corresponding to the message it received.

In statically–typed languages the subtyping mechanism is tied to an inheritance hierarchy. In dynamically–typed languages there is no need to use inheritance in order to achieve subtyping since any message can be sent to any object. Dynamic typing offers greater flexibility whereas static typing offers greater safety. The fact that most mainstream languages — including C++, the most widely used, and the currently popular Java — are statically typed show that language designers are not willing to give up safety in order to gain greater flexibility.

Class–Based versus Object–Based Class–based languages, such as C++, Java and Eiffel, are organised in terms of descriptions of sets of similar objects. Object–based languages, such as Self [Chambers et al., 1991] and Cecil [Chambers, 1997], on the other hand are organised around descriptions of individual objects. This has a strong impact on the meaning of inheritance in each category of language. In class–based languages inheritance occurs at the class–level, the subclass inherits the characteristics of the superclass. In object–based languages, one object inherits its characteristics from another object (the terminology used is that one object *delegates* some of its responsibilities to another object).

The object–based approach is more flexible — any object can delegate behaviour to any other object — but offers less structure. It becomes more difficult to model in terms of abstractions since every object can be different. The class–based approach is more structured but less flexible. Inheritance is defined at the class level rather than at the object level which means that inheritance relationships are fixed and cannot vary dynamically.

Design by Contract The design–by–contract [Meyer, 1992] style of software development sees the relationship between a class and its clients as a formal agreement. As well as providing a type, a class also provides a specification, or contract. Design–by–contract originated in the Eiffel language and Eiffel provides comprehensive support for it.

A contract is specified in terms of assertions. The types of assertion supported by Eiffel are: preconditions which state the properties that must hold when a method is invoked; postconditions which state the properties that must hold when a method returns; and invariants which must hold throughout the lifetime of an object. It is the obligation of the client to ensure that the precondition is true

before invoking a method, the server is then responsible for achieving the postcondition. Method behaviour is undefined if the precondition has not been met. For example, a counter class might have an invariant that the count is non–negative. The decrement method on such a class would have a precondition specifying that the current count must be greater than zero and a postcondition that specifies that the count will be decreased by one.

The design–by–contract principle has strong implications for inheritance. In order for subclass instances to be substitutable for superclass instances the subclass must maintain the contract of the superclass. In other words, a subclass inherits the contract of its superclass. The subclass can extend the contract as new functionality is added but it must always maintain the inherited part of the contract. This means that preconditions can only be weakened and postconditions can only be strengthened. For example, we could not create a positive counter subclass of the non–negative counter and override the precondition of decrement so that it requires the current value to be greater than one. This would strengthen the precondition but clients that only know about the superclass version of the method would not be aware of this restriction and so it cannot be introduced.

Overriding and Overloading There is an important distinction to be made between two forms of method inheritance that occur when a subclass method is defined with the same signature (name, number and type of arguments) as a superclass method. Overriding occurs when a method is replaced in the subclass by a specialised version. When the method is accessed on a subclass instance the specialised subclass version will be invoked. Overloading occurs when a specialised subclass version of a method is seen only by subclass clients. Superclass clients still see the original version. In other words the method that gets invoked depends on the type of the variable through which the subclass instance is accessed, rather than on the type of the instance itself.

Overriding is only possible when methods are dynamically bound; static binding leads to overloading. The distinction is important since C++, the most widely deployed object–oriented language, supports both forms of method inheritance with overloading being the default. In order for a method to be overridden in a subclass it must have been declared to be 'virtual' in the superclass. Figure 2.4 shows the inheritance of a virtual method, `time`, with overriding and a non–virtual method, `now`, with overloading (pseudo code is used in preference to the actual syntax of C++). When the overloaded `now` method is invoked the superclass version of the method gets called even though the receiver is a subclass instance. Note that the subclass versions of the methods are the same whether they are overriding or overloading superclass methods (or introducing new methods).

Overloading can also occur when the argument of an operation is specialised in a subtype: the method that is invoked at run–time will depend on the static type of the argument, rather that its dynamic type. This is what would happen in C++, in the restriction example where a coloured–point introduces an equality method that accepts only coloured–points, and an equality method for points is inherited from the superclass. When two coloured–points, held in supertype variables are compared, the point method for equality will be used.

Eiffel supports only overriding. Overloading, based on the type of the receiver or the arguments, is not supported for conceptual reasons: an object in a particular state should always behave in

```
class Clock
{
   virtual time(); // prints time in hh:mm format
   now(); // prints time in hh:mm format
}

class AccurateClock inherits Clock
{
    time(); // prints time in hh:mm:ss format
    now(); // prints time in hh:mm:ss format
}

Clock accurate = new AccurateClock("12:55:45");

accurate.time(); // prints time in hh:mm:ss format
accurate.now(); // prints time in hh:mm format
```

Figure 2.4: Overriding versus Overloading

the same manner when the same method is invoked. Java does not support overloading based on the type of the receiver, although it does support other forms of method overloading. In Java, if a subclass method has the same signature as a superclass method then overriding semantics always applies with respect to the receiver.

Method Extension A further distinction must be made between method replacement, as occurs with overriding, and method extension in which the superclass method is always invoked and in which the subclass can only define extensions that should also be executed. Method extension is significant since it guarantees that the behaviour of a superclass will also be executed by its subclasses. With method extension, a subclass may add to superclass behaviour but cannot replace it. This reduces the possibility of superclass clients being surprised when dealing with a subclass instance.

There are multiple ways of achieving method extension:

- **call to super** — In Smalltalk, method extension is achieved by inserting a 'call to super' into the body of the subclass method. When execution reaches that point the superclass method is called. This approach allows the subclass to determine the correct point for the superclass method to be invoked. This approach only works in single inheritance languages, a more complex approach is required in the case of multiple superclasses where the meaning of 'super' is ambiguous.

- **inner** — In Beta language [Madsen et al., 1994], the `INNER` keyword can be used to define the point at which a subclass extension to a method (should it exist) will be invoked. The superclass method is always invoked, when control reaches the `INNER` keyword the subclass extension to the method is executed, if it exists, and then control resumes from after the `INNER` keyword in the superclass. The Beta approach to method extension gives the superclass control over when subclass extensions will be executed.

- **before, after and around** — The Common Lisp Object System (CLOS) [Paepcke, 1993] provides support for method extension in the form of `before`, `after`, and `around` methods, each of which may extend the behaviour of an inherited method. A subclass can define methods

to be invoked before, after or around the inherited version of a method. A hierarchy of such methods can be built up with each subclass inheriting the extensions of its superclass and introducing its own.

Any before methods are called before the main method, from most specific to least specific. The after methods are called after the main method, from least specific to most specific. The return values of before and after methods are ignored. The around methods are run before all other methods with a 'call next' invocation being used to invoke the next around method and eventually the first before method or the main method; after execution of the after methods, execution returns to the around methods in reverse order.

- **assertions** — Although method extension is a structured way of building up a class there are occasions when method overriding must be used while still achieving the result of the superclass method. In Eiffel, a subclass inherits assertions from its superclass which must be maintained. The subclass must meet the obligations of the superclass version of the method as well as any additional subclass obligations. The superclass method need not be called directly but its outcome must be achieved in some way by the subclass. For example, the subclass may produce a more specialised result which meets superclass obligations and also stronger subclass requirements. This can be seen as conceptual method extension although an overriding language mechanism may be used to achieve it.

Subtyping and Subclassing Inheritance In mainstream object–oriented languages inheritance is typically an operation on classes and since classes introduce types, inheritance is also the subtyping mechanism. There is no reason why all inheritance relationships should always lead to subtyping.

In C++, it is possible to introduce inheritance relationships that do not lead to subtyping, this is referred to as private inheritance. Eiffel does not support inheritance without subtyping although methods may be *disinherited.* Strictly speaking this does not lead to a subtyping relationship. However, Eiffel supports a subtyping relationship for the operations that are inherited. To ensure type–safety complex checks must be carried out — calls to a disinherited method will cause a compile–time type error. In Java all inheritance relationships lead to subtyping. There is no way to implement non–subtyping inheritance.

It is widely acknowledged that the separation of subtyping and subclassing is a good thing [Porter, 1992] although support for this separation in mainstream languages is lacking [Powell, 1995]. Some less well–know languages such as Theta [Liskov et al., 1995] and POOL [America, 1989] separate types from classes and provide separate mechanisms for subtyping and subclassing.

Multiple Dispatch In C++, Eiffel and Java, methods are dynamically dispatched on the receiver only. The static types of objects passed as arguments to operations are used to determine which method to invoke. This means that, in C++ and Java, passing an object held in a supertype variable into a method may have a different result to passing in that same object held in a subtype variable. Eiffel avoids this problem by avoiding overloaded methods as described above. In order

to get a method to behave in a particular way based on the dynamic type of an object (that is, its most specific type) multiple dispatching is required. The modem connection scenario, in which a proprietary protocol will be used when both modems support it and a generic protocol will be used otherwise, requires multiple dispatch.

In statically–typed, single–dispatching languages, multiple dispatch can be simulated with double–dispatch (or sequential dispatch if more than two arguments contribute to method selection). In the Modem example a `connect` method would be invoked on the modem initiating the connection, that modem would then invoke a suitable method on the modem that is being dialed, the chosen method depends on the type of the receiver so that in the second dispatch the types of both arguments are effectively known. This approach is shown in Figure 2.5. If we invoke the `connect` method on an X–branded modem, it will then call the `xConnect` method on its argument. If the original argument, now the receiver of the second dispatch, is also an X–branded modem then it will use the brand–specific protocol, if on the other hand, the argument is a Y–branded modem then it will invoke the generic connection method `genericConnect`. Methods that dispatch on the dynamic types of arguments (other that the receiver) are referred to as multimethods. Multimethods are widely supported in Lisp–based languages such as CLOS.

```
class Modem
{
    connect();  // abstract method to be overridden

    xConnect(); // abstract method to be overridden
    yConnect(); // abstract method to be overridden

    genericConnect(); // generic connection protocol
}

class XModem
{
    connect();  // calls xConnect

    xConnect(); // implements X-specific protocol
    yConnect(); // invokes generic connect
}


class YModem
{
    connect();  // calls yConnect

    yConnect(); // implements Y-specific protocol
    xConnect(); // invokes generic connect
}
```

Figure 2.5: Double Dispatch

Inheritance and Genericity Genericity, like inheritance, allows families of related abstractions to share common code. A generic class (or parameterised class or template) is created which omits certain types from its definition. The generic class is instantiated by providing the missing types as parameters. Genericity can be unbounded, in which case any type can be used to instantiate the generic class, or bounded, in which case the parameter has an associated type and the type parameter must be a subtype of that type.

As an example of unbounded genericity we could have a generic list class `List[X]` where `X` is a type parameter. We can instantiate the generic list with a `String` type to get the class `List[String]`. Generic classes are not classes themselves, a generic class must be instantiated to provide a class and associated type.

If on the other hand, we want to create a generic sortable list where the type parameter must offer a comparison method, then we can use bounded genericity to define a generic class `SortedList[X < Comparable]`. We can then instantiate the sorted list with the type `String`, provided that `String` is a subtype of `Comparable`, to get the class `SortedList[String]`.

Inheritance and genericity have much in common as discussed by Meyer in [Meyer, 1986]: they both allow 'the definition of flexible software elements amenable to extension, reuse and combination'. In his classic paper 'Genericity versus inheritance' [Meyer, 1986] Meyer compares inheritance and genericity and shows that inheritance can simulate much of the behaviour of genericity whereas genericity cannot simulate inheritance. Inheritance is the more powerful concept but it cannot simulate unbounded genericity. Meyer supports bounded genericity via the inheritance hierarchy - this combination of inheritance and genericity supports the definition of binary methods as discussed in the following section.

Subtyping and Matching The observation that subclasses do not always lead to subtypes, as with the binary equality method in the point/coloured point example, has led researchers to try and find an alternative to inclusion polymorphism that supports the inheritance of binary methods and can still be statically type checked.

In Eiffel a 'like' construct is introduced to support the declaration of types that are 'anchored' to each other. This supports the specification of relationships such as the point/coloured point example discussed earlier, by declaring the type of the argument to the equality method to be 'like Current', that is, the same as the type of the receiver. The Eiffel type system can statically type check such programs, ensuring that the equality method can only be invoked on two objects with the same dynamic type (early versions of the Eiffel type system were shown to be unsound but recent versions can statically achieve type safety). The Eiffel approach requires complex type checking and is not considered an adequate solution by some. Under the Eiffel type system the addition of new code can invalidate existing code — the Eiffel type system does not support modular type checking: checking that individual classes are type–safe is not sufficient, system–wide type checks are also required.

A related type system is that suggested in [Palsberg and Schwartzbach, 1990]. Instead of using complex type checking as Eiffel does, the suggested type system relies on run–time type checking for a certain class of constructs (analogous to those for which Eiffel requires system–wide type checks). Again, this solution is not considered satisfactory by some language designers due to the value placed on static type checking.

In response to this problem a number of type systems have been proposed (see [Bruce, Cardelli, Castagna, The Hopkins Object Group, Leavens and Pierce, 1995] for a summary) that overcome this problem. One such approach [Bruce, Schuett and van Gent, 1995] refers to the resulting relationship

as 'matching' and the term 'matching' will be used to refer to relationships of this kind throughout this thesis (there are small technical differences between the approaches but they are not significant on a conceptual level). As yet, matching has not made it into a mainstream language.

Subtyping is said to be contravariant because argument types vary in the opposite direction to the subtype (the subtype is more specialised whereas argument types must become more general). Matching is said to be covariant based on converse reasoning. It has been shown that covariance and contravariance can coexist in a type system [Castagna, 1995]. This means that languages could choose to offer both subtyping and matching.

Note that although matching is related to genericity, both involve parameterised classes, the two constructs are not equivalent. As discussed in [Shang, 1994], genericity does not support dynamic covariance–based polymorphism.

Multiple Inheritance Multiple inheritance occurs when a subclass directly inherits from multiple superclasses. The subclass inherits the properties of all superclasses. For example, we might have a postgraduate teaching assistant inheriting from both Student and Staff superclasses.

If multiple inheritance is supported then a conflict resolution mechanism is required in the case that multiple properties with the same name are inherited. Both C++ and Eiffel support multiple inheritance and both languages provide conflict resolution mechanisms.

A special case of multiple inheritance occurs if a subclass inherits from the same superclass more than once along different paths. For example, both Student and Staff might inherit from University Member, leading to Postgrad TA inheriting from University Member twice, as shown in Figure 2.6. This is referred to as repeated inheritance. There are two ways of dealing with this situation, either each property is inherited multiple times or each property is inherited a single time. In the example shown in Figure 2.6 each property should be inherited only once, since a postgraduate teaching assistant is only a member of the university once. Both Eiffel and C++ allow these forms of repeated inheritance to be distinguished: Eiffel allows the distinction to be made on a per property basis whereas C++ only supports a per class distinction. The default for C++ is to duplicate the properties of repeated ancestors, for Eiffel the default is to inherit a single copy of the properties of repeated ancestors.

Java supports multiple inheritance only for interfaces (pure types that provide no implementation). This means that name clashes and repeated inheritance are not a problem. Repeated operations are considered to be one and the same operation and appear only once in an inheriting interface.

2.6 Conclusion

An intuitive understanding of inheritance comes easily to most software developers. However, a simple intuitive understanding of inheritance is not sufficient to make effective use of current inheritance mechanisms.

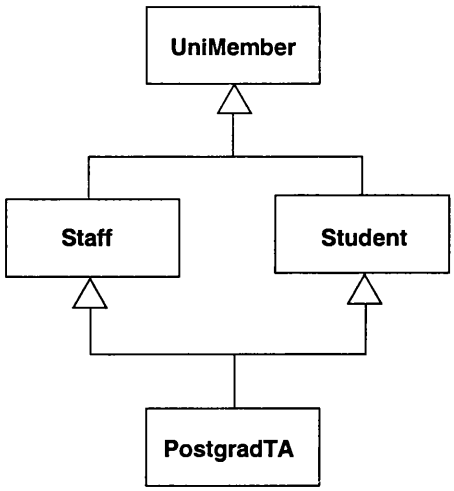

Figure 2.6: Repeated inheritance.

Using inheritance for its local and immediate effects can lead to problems such as spaghetti inheritance and the fragile base–class problem which only become apparent in a larger view of the system, and over time.

Further, even the local and immediate effects of inheritance are not as simple as they might seem. The variation in the details of inheritance mechanisms in statically–typed class–based languages such as C++, Java, and Eiffel is indicative of the complexity of object–oriented inheritance. The need to consider the semantics of inheritance in the face of multiple dispatch and matching further illustrates the inadequacy of a simple, intuitive understanding of inheritance.

The variety of advantages of inheritance and definitions of inheritance shows that a single inheritance relationship needs to be either general to the point of being useless in precise modelling or overloaded to the point of extreme complexity — the latter being the current situation.

Chapter 3

Structured Inheritance Relationships (SIRs)

In the previous chapter we saw that inheritance is a powerful mechanism with many advantages. We also saw that the flexibility of inheritance has its downside — complexity.

There is one form of inheritance that is widely accepted as the 'ideal' use of inheritance, it produces relationships that pass the is–a test and support inclusion polymorphism. Some authors go so far as to say that this should be the *only* use of inheritance [Rumbaugh, 1993]. While not advocating such a restrictive approach we recognise that mixing is–a relationships with other relationships under the umbrella term of inheritance is far from ideal. An improved situation would see the software developer with a number of well–defined relationships rather than a single overloaded relationship.

In this chapter we introduce five distinct relationships to replace the confusion of modelling concepts currently known as 'inheritance'. The term structured inheritance relationship (SIR) is used to describe these specialised forms of inheritance. The term 'structured' is used here to mean controlled or disciplined as opposed to ad–hoc. Structured inheritance leads to a clean architecture rather than unstructured spaghetti inheritance hierarchies.

Other authors have previously classified applications of inheritance. Both Budd [Budd, 1997] and Meyer [Meyer, 1996] have provided descriptions of common uses of inheritance in object–oriented programming languages but neither claims that their list is exhaustive. The descriptions of the five SIRs presented here are not simply an alternative classification to those presented by Meyer and Budd. Rather than simply attempting to explain existing uses of inheritance mechanisms in conceptual terms we take a step back and consider the conceptual relationships that inheritance mechanisms should be able to support. We take current usage of inheritance mechanisms as a starting point for identifying the conceptual relationships behind applications of inheritance, but we then develop natural extensions of these concepts at the modelling level where the constraints of particular inheritance mechanisms are not present.

This higher–level approach supports the consideration of inheritance as a modelling construct rather than being subject to the specifics of the inheritance mechanisms currently available in programming languages (as we saw in the previous chapter, the functionality offered by such mechanisms varies considerably). Importantly, the relationships are not considered in isolation. They each play a rôle within a larger framework.

We propose that the term 'inheritance' be replaced, within object–oriented modelling, by the more specialised terms view, variant, construction, specialisation and evolution corresponding to the five SIRs described in this chapter.

3.1 Context and Scope

In the past, software modelling techniques were mainly concerned with starting from nothing and building a system that fulfilled a predefined set of requirements. Much advice on the use of inheritance assumes such a scenario. This is a long way from the process by which much software is currently developed. The following factors characterise modern software development:

1. Systems are large; they are often not understood in detail by any individual.
2. Systems are not built in a single pass, software development is an iterative process.
3. The full requirements are rarely known or completely understood in advance.
4. System development does not end when the original requirements have been fulfilled. In successfully deployed systems new requirements will be added and existing requirements may be modified in the light of practical experience.
5. Systems are expected to reuse existing software components to decrease development costs and to reduce maintenance time.
6. Systems are expected to produce components that can be reused by other systems.

The relationships presented in this chapter are intended to support the development of software systems within the context of modern software development. This means that we will consider advantages and disadvantages in–the–large, rather than focusing on local and immediate benefits.

It is also necessary at this point, to clarify what is meant by the term 'inheritance'. In the previous chapter the factor common to all definitions of inheritance was found to be that the subclass shares the characteristics of the superclass. This is a useful intuitive definition but a more detailed definition of inheritance is required to specify the scope of inheritance for this analysis. Firstly, it is important that the subclass actually possesses the characteristics it inherits rather than simply having access to them. This allows the subclass to (optionally) offer inherited operations directly to its clients when this is conceptually appropriate. This is one area where the functionality offered by inheritance differs from that offered by clientship. It also supports overriding where it is permitted.

If the subclass does offer compatible versions of all superclass operations then a subtyping relationship may result. We do not exclude the possibility that superclass methods may be offered to subclass clients without an accompanying subtyping relationship. For maximum flexibility, both notions should be accommodated. A further important issue is that inheritance must be explicitly specified. Just because two entities are similar, it does not mean that there is an inheritance relationship between them. Inheritance must be explicitly stated. Additionally, inheritance is not just an operator that takes a superclass and outputs a subclass. A relationship between the superclass and subclass remains. If we modify the characteristics of the superclass, the characteristics of the subclass will also be changed. With these issues in mind, we propose the following definition of inheritance:

DEFN 1 **inheritance** Inheritance is a stated, continuing, relationship between two abstractions in which the characteristics of the first, the superclass, are also characteristics of the second, the subclass.

This definition does not say which characteristics of the superclass are inherited by the subclass and therefore permits specialised forms of inheritance in which different sets of characteristics are inherited. In addition, the definition only describes the characteristic–sharing aspect of inheritance, specific forms of inheritance may extend this definition with appropriate traits (such as subtyping). Note that the inheriting abstraction has no obligation to reveal inherited characteristics to its clients. Characteristics that were external (that is, visible to clients) in a superclass may be hidden in the subclass. The subclass must have those characteristics however, they cannot be 'disinherited' although they can be hidden by subclass operations. This definition encompasses the mainstream understanding of inheritance and supports the extension of that understanding.

Scope of Inheritance Definition

Inheritance, as defined above, is a relationship between two abstractions. This means that the following relationships fall outside the scope of this model:

- **genericity** — the relationship between a class template and a class, or a type generator and a type. Genericity creates multiple abstractions from a 'cookie cutter'. We are interested in relationships between abstractions; genericity operates at a level above this.
- **instance–of** — the relationship between an object and its class or type. The instance–of relationship operates at a level below inheritance.

Although these relationships can be styled 'is–a' relationships and are important relationships in their own right, they do not fall within the definition of inheritance used in this thesis. Similarly, we do not consider relationships between two templates (parameterized types or classes) or between two instances (objects), except where those relationships are a side–effect of inheritance.

Note that matching occurs when type is considered to be one of the characteristics that can be conformantly varied in an inheritance relationship. We do not explicitly consider matching in this

chapter, which is concerned with conceptual relationships. However, it is considered in the following chapter where a detailed model of inheritance is developed.

Encapsulation

Some inheritance relationships introduce strong coupling between the abstractions concerned. The SIR model does not concern itself with whether a subclass should be granted access to the internals of another class or whether it should have access only to its public interface. We assume that a separate mechanism is responsible for encapsulation. This rôle is ideally suited to module mechanisms as found in languages such as Ada [ANSI, 1983] and Oberon–2 [Mossenbock, 1993]. In the SIR model, inheritance comes in to play once access has been granted to certain operations. Inheritance must not abuse any access it has been granted — it must maintain the integrity of abstractions even if it has the power to do otherwise.

3.2 Variant

We now introduce the first of the five SIRs: variant. A sans serif font will be used to distinguish references to the variant relationship from other uses of the term variant.

The variant relationship is based on the observation that inheritance allows operations to be introduced in a superclass and then implemented, as methods, in a subclass. One reason for introducing an operation but not implementing it at the same time is to permit alternate implementations of that operation. Sometimes a subclass simply implements the operations it inherits from a superclass without introducing any new public methods. In such cases the instances of all such subclasses can be fully manipulated via the superclass interface because the subclass does not extend the interface. We will refer to such subclasses as *variants*. The relationship between a less complete and a more complete abstraction is the variant relationship.

Just because objects present the same interface to the world, and cannot be distinguished by clients, it does not follow that they will have the same internal behaviours. Objects that are classified in the same way need not be behaviourally identical. Examples of variants include:

1. Cartesian and polar representations of points.
2. Time efficient and space efficient implementations of data structures.
3. A local disk and a remote–mounted disk on a Unix machine.
4. Immutable empty lists and non–empty lists.

With the variant relationship we do not introduce new categories of object (types) in order to provide alternate realizations of objects that fall within that category. This reduces the overall number of high–level abstractions within a system, making it simpler to understand. The distinctions between

different variants within a system need only be understood when they are instantiated, after that all such objects are treated the same.

In order to discuss the variant relationship precisely, we introduce terminology for the superclass and subclass in a variant relationship:

DEFN 2 **supervariant, subvariant** A class that provides methods corresponding to interfaces defined in a superclass is referred to as a subvariant, the superclass in such a relationship is referred to as the supervariant.

It is also useful to be able to refer to a set of variants with the same supervariant:

DEFN 3 **variant family** The set of subvariants of a particular supervariant is called a **variant family**.

We will also refer to the variants of a type, meaning all of the classes that implement, via a variant relationship, a particular type.

3.2.1 Using Variant as a Modelling Tool

Suppose we are modelling a mail order billing system in which various discounts can apply to an order. We have 3-for-2, buy A get B free, spend over amount C and get D% off, and whatever the sales people come up with in the future. Although each of these discounts has different information associated with it and will calculate money off in a different way, they will all be *used* in exactly the same way. This indicates that variants should be used to model the differences. Figure 3.1 shows the resulting class hierarchy with the **var** qualifier being used to indicate that the inheritance relationship is variant. The Discount abstraction contains an `apply` operation that applies a discount to an Order (we assume that order objects maintain information on the items and quantities ordered and the total value of the order).

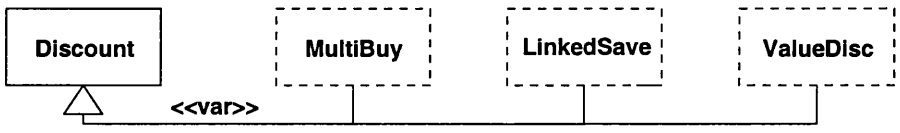

Figure 3.1: `Discount` base abstraction and its variants.

It is conceivable that the developer writing the ordering code, which applies the discounts, does not have any knowledge of the kinds of discount that will be present in the system. This is a useful form of abstraction. Since clients only know about the Discount type new variants can be added without needing to change client code. In fact we can modify the behaviour of a system considerably by introducing new variants, the only changes to classes outside the variant are those needed to instantiate instances of the new variant.

Introducing the variant relationship to handle this situation leaves the main classification hierarchy for abstractions that are of direct use to clients. We do not pollute the main type hierarchy with

types that add no value for clients (subvariants offer no properties in addition to those offered by supervariants).

We also simplify the task of understanding an inheritance hierarchy: where a variant family exists there is no need to consider the individual variants when attempting to understand a system, they are only of interest when *creating* variant instances or designing new variants. Rather than being faced with a plethora of specific discount classes that only differ in their detail we can stop at the Discount abstraction satisfied that we have all important information about that abstraction.

Traditionally, all inheritance relationships are of equal importance. Now we have a situation where variant relationships operate at a finer granularity than *is–a* relationships thus allowing hierarchies to be understood in stages. This is a welcome step towards making inheritance hierarchies more understandable.

3.2.2 Variant Subrelationships

All variant relationships support the modelling of behavioural differences that are not visible to clients. However, not all variant relationships lead to the same kind of behavioural differences. We now summarize the kinds of variant relationship that are possible.

Alternate Implementations Alternate implementations are equivalent implementations which are fully interchangeable, that is, there are no criteria for choosing between the implementations.

Two variant implementations of a supervariant are equivalent if, and only if, sending the same sequence of messages with identical arguments (including the same initialization values) to an instance of each always gives the same results.

Constructors may differ for alternate implementations but they lead to exactly the same set of initial states so that either could be used in any situation. Clearly, we never have the need for alternate implementations of the same abstraction. In general, if two such implementations are discovered one should be discarded in favour of the other.

Non–Functionally Distinguishable Variants Non–functionally distinguishable variants are equivalent implementations with different non–functional selection criteria.

Non–functionally distinguishable variants allow us to model behavioural differences that do not affect the actual result obtained but affect the way in which it is obtained. For instance, we may wish to offer variant sorting classes to handle data that is likely to have certain characteristics (such as nearly sorted data versus random data); the same result will be achieved in each case but the time efficiencies and space efficiencies will be different.

The selection criteria for non–functionally distinguishable variants should be included in the specifications of variant constructors for use by clients. Non–functionally distinguishable variants may be initialised using different constructors or they may be created by a factory object that uses contextual information to determine the correct variant for a particular instance.

Partitioning Variants Partitioning variants subdivide the value space of a base abstraction into disjoint subsets. The initial conditions of an object determine the correct variant and therefore the subsequent behaviour of the object.

The order discount example is an example of partitioning variants. The correct variant is chosen when an object is initialised and the discount object behaves appropriately throughout its lifetime.

3.3 View

The next SIR is based on a particular use of multiple inheritance. Multiple inheritance allows an object to be accessed via different types: its direct type, and each of its supertypes. In this way it is possible to develop different interfaces to the same object which are appropriate to different kinds of client. In some cases, such interfaces are not intrinsic to the abstraction being described, they are only relevant from the perspective of certain clients — this is the SIR view relationship. For example, a cinema might view a student as a discount–price customer, this is not an intrinsic aspect of being a student, but it is how the cinema sees a student.

It is well known that classification is subjective, you cannot just classify things — you must classify them according to some criteria. As systems get larger and more complex it is increasingly difficult to describe any abstraction in a way that suits all of its clients (present and future). Each client will have its own requirements for an abstraction and will wish to see that abstraction under its own subjective interpretation.

In [McGregor and Korson, 1993] the authors state that 'structures that model the natural classifications of a concept will be more robust than those that model one possible solution to a problem'. Combining the natural classification of abstractions within a domain area with subjective classifications complicates the classification hierarchy. The view relationship allows subjective classifications to be modelled outside of the main classification hierarchy. Examples of different views of an abstraction include:

1. View of a person as a student, parent, employee, patient, etc.
2. Read–only and write access to a file.
3. Financial view of a house as an asset and personal view of a house as a home.
4. Views of various things found in supermarkets (apples, milk, bread, etc) as shopping list entries.

Different views of an abstraction may be more or less restrictive than each other or simply different. Views often correspond to the rôles that an abstraction plays during its interactions with other abstractions. The view relationship allows a single abstraction to cater to a number of clients all with differing requirements. It also allows us to separate the interfaces required by different clients to prevent muddled programming using elements of different rôles by creating a separate type for each interface. A view instance is an object in its own right and can have behaviour and state associated with it.

A view object provides an alternate way of interacting with an abstraction. Identity is maintained across views so that different views of the same object will inherit the identity of the target — they present the same underlying entity. Since the original interface to a target object is maintained, the view must must not cause the target object to violate its contract with existing clients.

We introduce some terminology to describe the subclass and superclass in a view relationship.

DEFN 4 **view, target** A **view** is an abstraction that is based on another abstraction, the **target** of the view, and provides the interpretation of that abstraction required by a particular kind of client of the target.

3.3.1 Using View as a Modelling Tool

A classic example of the view relationship occurs when modelling a buffer. Buffers have two specific sets of clients: producers and consumers. For producers a buffer acts as a sink and for consumers it acts as a source. This relationship is shown in Figure 3.2. In this sort of view relationship the views allow an abstraction to play different *rôles*. The consumer view of the buffer sees the buffer in its source rôle and the producer view of the buffer sees the buffer in its sink rôle.

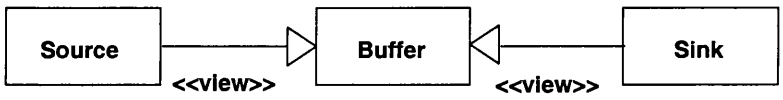

Figure 3.2: Source and Sink views of Buffer.

A useful aspect of this approach is that clients such as the producer can be written independently from the buffer in terms of a sink rôle, a view then allows the buffer to play the rôle of a sink. Other views may be created to allow other abstractions to play the same rôle. Clients have access only to the operations that are appropriate to the current rôle of an object; a client treating a buffer as a sink cannot access the get operation. Having separate interfaces for separate rôles allows developers to express their intent for a particular relationship. Even if the same client object acts as both producer and consumer for the same buffer the client can access that buffer via different interfaces, clearly specifying intent at each point where the buffer is used.

When views are implemented using inheritance, it is not possible to have multiple instances of the same view, each with associated state. For example, suppose we have several producers viewing the same buffer as a sink, it would be useful to keep track of the number of items produced by a particular producer for a particular buffer. We can add behaviour to the put method of the sink view so that it records this information, but if inheritance is used then there is only one instance of the sink view per buffer, the put method cannot determine which producer put a particular item and cannot keep track of production on a per–producer basis.

A natural extension of the view concept is to allow views to be instantiated on a per–client basis so that each view instance has its own state. In this case, the put method can record how many items

were produced via that particular view. The view relationship supports this behaviour which will be discussed in more detail in the following chapter.

3.3.2 View Subrelationships

In the view relationship we have built up a powerful technique which allows an abstraction to play different rôles in different contexts. In practice, most views will be fairly simple. In this section we describe possible view relationships starting with the simplest.

Alternate Interface A view may simply provide an alternate interface to target objects that is more appropriate to some clients than the interface provided by the default view. The alternate interface may contain more operations than the original, less operations or just different operations.

Alternate interfaces are useful when an abstraction provides functionality that needs to be presented to different clients in different ways. The differences may be as simple as name changes in operations, or changes to the order of arguments. Alternate interfaces are also useful for allowing certain clients to access only a subset of the behaviour of an abstraction. Alternate interfaces differ from other views in that they do not have any state associated with them on either a per view or per client basis.

The sink and source views of a buffer provide alternate interfaces to producer and consumer clients.

Subjective Behaviour A view may add behaviours that cannot be implemented in the target itself since they are dependent on some subjective element that is not a part of the target abstraction.

The iterator abstractions commonly used in object–oriented programming provide an example of subjective behaviour. Naive attempts to provide iterator functionality add iteration operations directly to an abstraction, for example by adding next and reset operations to a List abstraction. This fails because there is no notion of the current element of a list and if there were there would only be one current element. It would not be appropriate for a list to maintain a whole list of current elements, one for each client.

If an iterator is implemented as a view then it can have a current element since a new view instance can be created for each client requiring an iterator.

Mapping A view may map elements of one abstraction to elements of another abstraction. The view provides behavioural substitutability with the required abstraction.

An example of this would be viewing a pair as a triple. Note that there are several possible variants of this view using different abstraction mappings to map two of the elements of the triple to a pair, or even the minimum and maximum element of the triple or some other mapping. We could also provide an inverse mapping if we wished to view a pair as a triple, in this case we could insert an additional default value in the third slot.

The view `PairAsTriple` is a variant of the `Triple` abstraction so pairs viewed as triples are treated (and must behave) exactly as other variants of the `Triple` abstraction which may store their values

directly. As always, modifications to the target made via one view of an abstraction will be visible by other views so it is necessary for the view to maintain the contract of the target abstraction.

3.4 Evolution

Systems that are written in a single iteration by a single person and never modified are a rare occurrence in modern software development. A typical system undergoes several iterations with different developers, or teams of developers, working on parts of the system. The first version of a successful system is almost never the last version. In addition, abstractions designed for one system may be reused in another system.

In this environment, any specification of an abstraction is likely to be found lacking at some point. Details that were missed or abstracted away as unimportant when designing a system may become a crucial aspect of an abstraction at some point in its lifecycle. The environment in which the abstraction operates may change or the requirements for the abstraction in the current system may change. Thus, the description of an abstraction may need to evolve over time.

Inheritance is often used to model such changes in the description of an abstraction. With inheritance we get a clear distinction between the original abstraction and the new version of it. Examples of evolution include:

1. Adding date of birth information to a person class.
2. Updating a taxation class to reflect changes in the law.
3. Modifying the implementation of a matrix class to make use of parallelism.
4. Evolving an HTML parser so that it can cope with a new version of the HTML standard.
5. Improving a dictionary class with a faster look–up algorithm.

In each of these cases, the underlying abstraction captures the same idea before and after evolution, only the details have changed. Evolution allows the implementation of a complex abstraction to be built up over time.

The following terminology will be used when discussing evolution:

DEFN 5 **original abstraction, resulting abstraction** The term **original abstraction** refers to an abstraction before evolution has taken place; **resulting abstraction** refers to the abstraction after evolution has taken place.

DEFN 6 **adaptation** An **adaptation** of an original abstraction refers to the changes required to evolve an abstraction to suit a new understanding of the concept that it models.

Note that evolution differs from other forms of inheritance in that the original abstraction and the new abstraction are not intended for use within the same system. Evolution has typically been

considered to be outside the scope of a design, it is the relationship between successive designs. Although this may be satisfactory for single systems, when abstractions are used by multiple systems it is important to bring the evolution relationship into the design and to allow different systems to evolve abstractions in different ways.

Evolution is either conformant, in which case existing clients will be satisfied by the resulting abstraction, or the clients of the abstraction must also be included in the unit of evolution. An example of the latter occurs when Java methods are marked as deprecated, such methods have become unnecessary due to the evolution of the abstraction to which they are attached. Clients are given time to evolve while the method is marked deprecated; after a period of time the method will be removed. Since all clients should have been updated while the method was marked deprecated the result is that the abstraction and its clients have been evolved as a unit.

3.4.1 Using Evolution as a Modelling Tool

Consider a drawing application developed to run on black and white terminals. The application has a Point class to represent 2D points that can be drawn on the screen. The application must now move to colour terminals. Under this new set of requirements, the interpretation of the point abstraction changes: now points have a colour.

We could modify the Point abstraction itself but this is not appropriate if there are other applications using the same class and it may not even be possible if the original source code is not available. (If there are no reasons to prevent the modification of the original point class then modifying the source code is a valid implementation of an evolution relationship identified at the design level.)

We can create a Colour Point subclass which adds colour to the Point class (as shown in Figure 3.3) and adapts point operations to their new environment where necessary. Coloured point instances will be substitutable for point instances throughout the application due to subtype polymorphism. Note that we are not distinguishing between points that have a colour and point that do not (although obviously there are contexts where this relationship would be appropriate), we are saying that the appropriate interpretation of the point abstraction needs to be different in a new system.

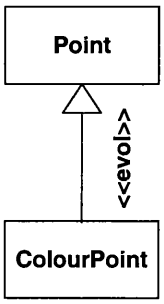

Figure 3.3: Evolution of Point to ColouredPoint.

3.4.2 Evolution Subrelationships

First of all we have two categories of evolution into which all of the other sub–relationships fall:

- **required adaptation** — A *required adaptation* must be included in any interpretation of an original abstraction.
- **optional adaptation** — An *optional adaptation* for a major abstraction may be used to configure an abstraction for use in a particular context; there is no requirement for optional adaptations to be used.

Over time required adaptations may be combined into the original abstraction. This is especially appropriate if the abstraction is still under development, in this case the required adaptations should be merged into the original abstraction. Other adaptations will never be combined into the original abstraction even though they are required, since separating them out serves another purpose such as simplicity.

During system development, required adaptations document the evolution of an abstraction. Once the abstraction has become stable this information becomes less important, the focus shifts from how the abstraction developed to the resulting abstraction.

Within a particular domain we may wish to provide a framework for the specification of abstractions for use in a particular application. For example, we may require that exactly one adaptation from a given set is used in any abstraction. The term 'required adaptation' may therefore refer to a general adaptation which itself has optional adaptations, exactly one of which must be selected for a given system.

Optional adaptations allow an abstraction to be tailored to suit the requirements of a particular system. Systems that do not require the additional functionality are not forced to carry the additional costs of providing it (both in terms of run–time overhead and conceptual simplicity).

Evolution can also be characterised in terms of the nature of the adaptation that takes place.

Implementation Modification An adaptation overrides all or part of an existing implementation.

Implementation modification may be performed to fix bugs in third party libraries. In such cases direct modification of the source code may not be possible, and in any case is not desirable since the modifications cannot be reused when a new version of the library is released.

Implementation modification may also be used to make non bug–fixing modifications such as changing an implementation to rely on a different third party library or implementing an improved algorithm.

Addition of Services In an addition of services relationship new operations that are applicable to the abstraction are added.

Addition of services is purely for convenience. It allows us to implement operations within a class rather than outside it. An implementation within an object is often more efficient since it has access to internal information.

Addition of services does not normally introduce new behaviour to an object, the services added can be 'explained' in terms of existing services. The exception to this is when an oversight was made in the original abstraction, and behaviours that should have been possible (that is, the behaviours are allowed by the specification) were not provided. In this case we cannot simulate addition of services by writing operations that are external to the class.

Addition of Properties In an **addition of properties** relation we augment the definition of an abstraction with information that was not considered important when the abstraction was created but is now a necessary part of the description of the abstraction. New state may be added to the abstraction along with new operations to manipulate it.

The difference between the addition of properties relationship and an is–a relationship is that we are not describing a new, more specialised abstraction, instead we are saying that our original description of an abstraction was not detailed enough. The evolution of points to include colour is an example of addition of properties.

An is–a relationship between two concrete classes usually distinguishes between the instances of the abstraction that do have additional properties and those that do not. In an evolution relationship when we add properties we are saying that *all* instances of the abstraction have the additional properties, it is just that until now clients had not been interested in them so they were abstracted away.

3.5 Construction

Inheritance also allows a subclass to use inherited methods to implement its own interface. For example, a list might be implemented in terms of an array. The SIR construction relationship captures this use of inheritance. Construction[1] allows the implementation of one abstraction to include the implementation of another, previously developed, abstraction. The operations inherited in a construction relationship may or may not be made available to the subclass client, and even if they are, there is no substitutability relationship. Examples of the construction relationship include:

1. Implementing a list in terms of an array.
2. A person abstraction may include the behaviour of a name abstraction where name is an intrinsic property of people.
3. The name abstraction may, in turn, be based on a string abstraction.
4. Various graphical application windows may be constructed using the same menu abstraction.

[1]The term construction is taken from Budd [Budd, 1997]. Other names for relationships similar to the one described here include 'code inheritance' and 'inheritance for reuse'.

As well as providing reuse of code (and the effort taken to design, implement and test that code) we also gain a further benefit: where objects appear to have similar behaviours (as in the graphical applications example) those objects really do share the same behaviour. This consistency can be of enormous benefit to both modifiers of a system and end–users of a system.

Code–reuse based on the inheritance mechanism is defined at the class–level: in most object–oriented languages this means that an abstraction cannot change its implementation at run–time. A change of implementation at run–time would mean that objects would need to change class and this cannot be accommodated in statically–typed languages. By defining construction at the class level, but instantiating it at the instance level, we can make construction more powerful than inheritance–based software reuse. The issues concerning having construction introduce an object–level relationship are discussed in the following chapter where a detailed model of the SIRs is developed.

As with the other inheritance relationships we introduce terminology related to the construction relationship.

DEFN 7 **construction unit (CU)** A construction unit is an implementation class that is used (or is intended for use) in a construction relationship.

The terms superclass and subclass will be used to refer to the abstractions in a construction relationship.

3.5.1 Using Construction as a Modelling Tool

A frequently cited example of inheritance for code reuse is the implementation of a stack based upon an existing array class. This relationship is sometimes criticised since in most contexts it does not make sense to say that a stack is an array, only that a stack can be implemented using an array. A version of this appears in [Meyer, 1997] with an Arrayed Stack subclass of both Stack and Array.

A preferable way of modelling such a relationship is to consider Arrayed Stack as a variant of Stack that is in a construction relationship with the Array implementation; this model is given in Figure 3.4.

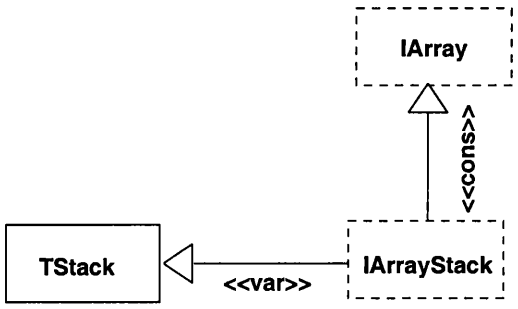

Figure 3.4: Using construction to provide an implementation of a Stack using an Array.

In this way we can use the existing Array implementation to construct the Stack class and there is a resulting variant relationship between the Arrayed Stack and the Array abstractions. This does not

introduce a subtyping relationship between stacks and arrays which is appropriate since we would not expect to use an array when a stack is required. The construction relationship allows us to model relationships in which the behaviour of one abstraction also applies to another abstraction, but in which substitutability is not necessarily appropriate.

Since construction operates at the object level, it is possible to change the superclass instance dynamically, so it would be possible to switch between array and linked–list implementations of stack at run–time if required.

3.5.2 Construction Subrelationships

Construction can be classified as follows:

Pass–Through Construction Pass–through construction occurs when all of the operations inherited from the superclass are passed–though the subclass directly to its clients. That is, no operation hiding takes place. Further operations may be added by the subclass.

Pass–through construction allows a subclass to provide all services provided by the superclass and some additional services. For example, it may be appropriate for a person abstraction to offer all operations related to the name abstraction (`name`, `surname`, `forenames`, `initials`, etc).

Construction–By–Difference The subclass may offer some operations unchanged and others not at all. New operations may be added by the subclass and may or may not be implemented in terms of superclass methods.

Construction–by–difference is a powerful form of construction that allows a subclass to be described in terms of how it differs from an existing implementation.

Internal Construction Internal construction allows a subclass to use superclass operations for implementing its own operations but does not offer those operations to subclass clients.

Internal construction allows one implementation to be written 'in terms of' another as in the stack and array example.

3.6 Specialisation

A pure view of the 'is–a' relationship is often obscured by the need to accommodate valid modelling relationships that do not lead to a specialisation relationship. Since we have already accounted for four major uses of inheritance that do not fit the normal interpretation of 'is–a' we have considerably more freedom in defining a specialisation relationship.

The four relationships described above can all be considered to be 'is–a' relationships if we use the term 'is–a' as it is used in natural language:

1. A variant 'is–an' example of its base. With the variant relationship we have separated out the cases where increased detail is not visible to clients of an abstraction.

2. A view of a target 'is–a' target instance looked at in a particular way. With the view relationship we have separated out cases where we are viewing the target via a mapping rather than simply ignoring certain details, and also the cases where a particular view of an abstraction does not apply to all instances. View is used when the characteristics of the subclass are not intrinsic to the abstraction to which they are applied.

3. An adaptation 'is–an' original abstraction in the light of new information. With the evolution relationship we have separated out the case where omitted information is added to an abstraction. We do not have objects with the additional properties and objects without those additional properties in the same system.

4. An abstraction constructed from an existing abstraction 'is–an' existing abstraction with certain differences. With the construction relationship we have separated out the case where the relationship is concerned with the implementation details of an abstraction rather than its external behaviour.

The ambiguity of using 'is-a' as the defining characteristic of inheritance is well known [Brachman, 1983]. With the specialisation relationship we are concerned with 'is–a' relationships that allow the same objects to be considered at different levels of abstraction (that is, in more or less detail). This means that the specialisation relationship supports the modelling of objects at different levels of abstraction, creating a hierarchy of types. As we move to higher levels of abstraction more objects are able to fulfill the class membership criteria since high level descriptions specify less about their members. Each client is able to access objects at a level that is suitable for its interactions with those objects. Examples of the specialisation relationship include:

1. Deciduous and evergreen subtypes of tree.

2. Positive, negative and zero subtypes of an immutable integer type.

3. Text editor and graphical editor subtypes of editor.

4. Child, Adult, OAP and Student subtypes of Customer.

The following terminology is used when discussing the specialisation relationship.

DEFN 8 **supertype/subtype** A subtype is a specialisation of a supertype. The subtype contains additional information that distinguishes instances of the subtype from instances of other subtypes of the supertype.

3.6.1 Using Specialisation as a Modelling Tool

Consider a university database system for keeping track of members of the university. Figure 3.5 shows a possible specialisation hierarchy for modelling university members.

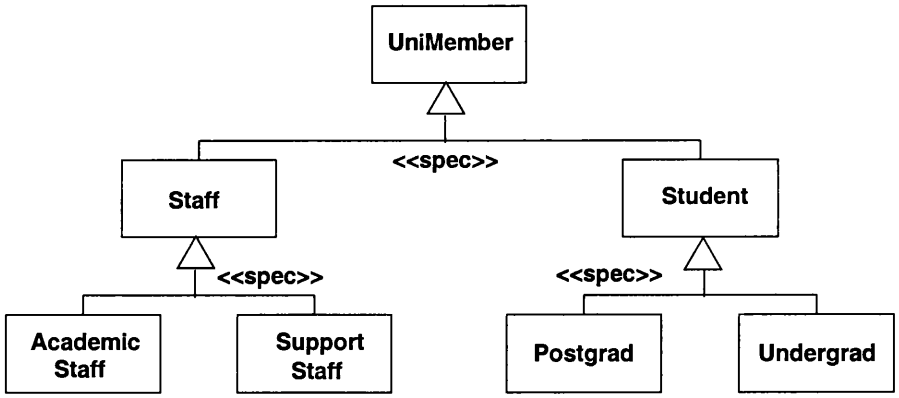

Figure 3.5: University Member Hierarchy

As we move down the hierarchy we can add more information to the abstractions, and thus more behaviour based on that information. For example, we may know that every university member has a name and contact details. Further down the hierarchy we acquire further information, students have a student number and are associated with a course of study. Staff have a payroll number and job title. Students are specialised further into undergraduates and postgraduates who will have a first degree (we could further subdivide this category into postgraduates on taught courses and research postgraduates).

Acquiring extra information about an abstraction allows us to provide behaviours based on that information. This may mean specialising existing behaviours or it may mean adding new behaviours such as creating a course list for an undergraduate student.

3.6.2 Specialisation Subrelationships

All specialisation relationships lead to an is–a relationship with full behavioural substitutability. Within this category of inheritance there are a number of variations.

Model Specialisation The subtype adds information to the abstract model inherited from the supertype.

As a result of model specialisation a further two forms of specialisation are possible (in practice these are usually combined into a single specialisation relationship rather than applied sequentially).

Behavioural Specialisation The subtype may specify that it requires less from its clients and/or that it provides more in return (this is the Design–by–Contract principle).

Specialised Extension Due to a model specialisation, additional operations, related to those inherited from the supertype, can be introduced. These operations do not apply to all supertype instances, they only apply to those with the additional properties specified in a model specialisation.

For example, we may have a supertype Book with a (specialised extension) subtype Collection representing books that are collections of contributions from various authors. The more general

book abstraction may have operations related to properties such as title, publisher, publication date and so on, the collection may add to this with operations related to the editor of the collection.

Note that when multiple inheritance is permitted, specialisation extension may occur through the combination of properties of two subtypes, each having an impact on the behaviour of the other.

Orthogonal Extension Orthogonal extension occurs when we introduce properties related to an aspect of an abstraction that is not a natural extension of the supertype. The added properties do not have any impact on the behaviour of operations that are inherited from the supertype.

For example, we may add operations concerned with printing to an Employee Record abstraction. These operations should have no effect on inherited operations.

3.7 Conclusion

In this chapter we have focused on whether or not a particular application of inheritance is a useful modelling technique, rather than whether it is a good use of current inheritance mechanisms. We have not been constrained by current inheritance mechanisms, this has led to powerful conceptual relationships that have been shown to exceed the modelling power of current inheritance mechanisms (such as that found in C++). Powerful techniques that have a strong conceptual foundation should certainly be part of the modeller's tool–kit.

We have developed a general definition of inheritance and introduced five SIRs that describe specialised forms of inheritance: specialisation, variant, view, construction and evolution. Specialisation replaces the confused notion of 'is–a' with a clearly defined conceptual relationship. This is possible since the other valid uses of inheritance are handled by the other SIRs. It is expected that specialisation will occur less often in systems developed within the SIR model than subtyping occurs in current systems. This is because specialisation does not occur as a side–effect of other forms of inheritance in the SIR model, as it does under current approaches.

Outside of the SIR model it is possible to build hierarchies around differences in implementation. For example, having different Printer abstractions to cope with different physical printers. The variant relationship is used for this purpose in the SIR model and this discourages unnecessary differences between types. The specialisation relationship will not be used to relate implementation classes with types that are identical, or with types that differ only slightly because a general interface was not developed. Evolution may also be used in the SIR model where specialisation might have been used in other systems. Rather than creating a new subtype which represents a fuller or system–specific understanding of the same concept, evolution would be used to enhance the existing description of the concept. For example, extending the Printer abstraction to cope with duplex printers can be handled with evolution.

Additionally, the view relationship will often be used in cases where specialisation might typically have been used. The view relationship should be used when one concept can play the rôle of another in certain situations. For example, we might use specialisation to relate the Vehicle and Car abstractions

but the view relationship to allow a Car to play the rôle of a Taxi. Although all Cars are vehicles, not all Cars are Taxis so there is no need to use specialisation for this second relationship. Similarly, specialisation does not coincide with construction as it would in models where there is a one–to–one correspondence between types and classes.

The introduction of five specialised inheritance relationships supports finer–grained modelling leading to systems that are more precisely specified. The resulting lack of ambiguity and confusion will lead to more understandable and maintainable systems. The SIR model leads to structured architectures rather than the spaghetti inheritance hierarchies that are currently prevalent.

We recommend that the term inheritance be confined to the description of language mechanisms and that the more specialised terminology introduced in this chapter should form the design vocabulary for discussing inheritance relationships. In the following chapter, we develop a complete model of design–level inheritance in which the five SIRs introduced in this chapter are the fundamental forms of inheritance which can be used to explain all current uses of inheritance.

Chapter 4

A New Model of Inheritance

In the previous chapter, five conceptually distinct inheritance relationships were introduced: specialisation, variant, view, construction and evolution. Each of these relationships was shown to be a powerful modelling tool in its own right. In this chapter we build a new model of inheritance in which the five SIRs are the fundamental building blocks. The five SIRs are described in detail including discussions of key decisions regarding the semantics of each relationship. The relationships are also contrasted with other SIRs, illustrating that the relationships are mutually orthogonal at a conceptual level.

4.1 Underlying Model

Before considering each of the SIRs in detail we develop the infrastructure that will provide the basis for all of the SIRs.

4.1.1 The SIR Notion of Types and Classes

A type describes the shared characteristics of a set of domain objects (the direct instances of the type and its subtypes). When the term type is used within the SIR model it is a shorthand for the full definition of a concept including its interface and its behavioural specification.

The SIR model separates the types and the classes that implement those types. The classes that implement a type are said to be variants of that type and are introduced using the variant relationship. However, the SIR model also permits inheritance relationships between implementation classes. So does this mean that, say, a construction relationship between two abstractions, is not based on the type of the inherited abstraction? In fact, it is preferable to consider this relationship in terms of the type of the superclass since a construction unit may introduce methods which should be accessible from the subclasses constructed from it. When a new construction unit is instantiated we must have a type by which to refer to it.

Rather than saying that the variant, construction and view relationships do not introduce new types we say that they introduce **secondary types**. Secondary types describe abstractions that are used in the realization of a system. The instances of secondary types do not necessarily correspond to domain concepts — they may be concerned with the realization of the system. The types related by specialisation, which are concerned with domain abstractions, are referred to as **primary types** when a distinction is necessary. Secondary types can only be accessed from within implementation classes, the SIR model does not permit the definitions of primary types to refer to secondary types.

A system modelled using the SIR approach has a top or system layer which permits only relationships between primary types (see Figure 4.1). Below that is the realization layer which permits relationships between secondary types and other primary and secondary types. This approach forces a clear separation between domain concepts that represent aspects of the application domain, and realization concepts that exist to support domain concepts. This is important since it reduces coupling at the domain level: primary types are only related by inheritance if there is a conceptual specialisation relationship between them, similarities in implementation are modelled at the realization level. (Note that both domain concepts and realization concepts are valid forms of abstraction that need to be modelled and designed. The term abstraction is not restricted to the description of domain concepts.)

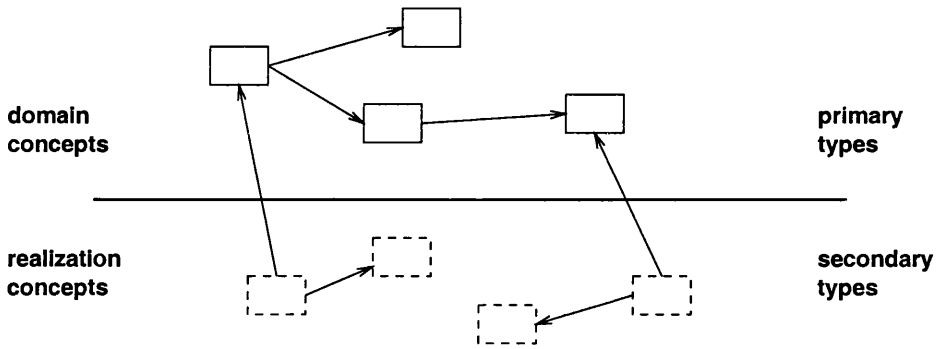

Figure 4.1: The domain layer and realization layer in the SIR model.

SIR Notion of an Abstract Class In programming languages where type and implementation are combined into a class concept, an abstract class is a class that has no direct instances. In the SIR model we must consider both abstract types and abstract implementation classes.

An abstract implementation class, in the SIR model, is incomplete and missing methods must be provided by a subclass. An abstract modifier on a type means that the type does not and *cannot* have any direct instances, that is, direct concrete variants are not permitted. The reason for this would be that the type is not sufficient to describe actual instances, further information, provided by subtypes, is required.

The SIR approach supports more fine–grained modelling: it is possible to model both incomplete concepts and incomplete implementations.

Genericity Unbounded genericity, in which the type parameter is not restricted to be a subtype of a particular type, is assumed to be available in the SIR model. A primary type or a class generated using a parameterised type or class is equivalent to any other type or class. Genericity does not introduce any type relationship between the types it generates. Unbounded genericity plays a rôle that is not met by inheritance (as discussed in [Meyer, 1986]).

4.1.2 Methods in the SIR Model

In Chapter 2 a number of different kinds of method and method inheritance were discussed. The following model of method inheritance is required in order to support the SIR model.

No Method Overloading Method overloading in object-oriented languages such as Java and C++ relies on the *static* type of arguments to distinguish between different versions of an overloaded method. This leads to problems when combined with subtyping (specialisation). If a method is overloaded for a subtype and a supertype argument, then it is possible for either method to be invoked when the argument is a subtype instance, depending on its static type. This leads to confusion even for experienced programmers and is therefore not supported by the SIR model.

C++ also allows static overloading based on the type of the receiver so that different methods may be called for the same method invocation on the same object depending on the type of the variable in which it is held. Meyer argues that static binding is not conceptually valid since the same method invocation on the same object can give different results [Meyer, 1988]. Overloading, based on the static type of the arguments or the receiver, makes the behaviour of systems difficult to predict and is therefore not used in the SIR model of inheritance. Although confusing, overloading of this kind does have a valid use: it enables the client to state that an object should be treated as subtype instance or, as a supertype instance. In certain circumstances this may be desirable. The SIR model provides the view relationship which allows a client to influence the treatment of an object so method overloading based on the type of the receiver is not required for this valid use.

Multimethods Dynamic dispatch on the type of the receiver of a method is a key feature of object–oriented systems. We argue that the same approach should hold for further arguments, in other words, multimethods should be supported. It is sensible to dispatch on the dynamic type of an object, wherever that object appears in a list of arguments. Therefore, the argument for not supporting method overloading based on the type of the receiver also extends to further arguments. The SIR model must therefore disallow multiple methods with the same name and number of arguments, or it must support multimethods.

Supporting multimethods raises the problem of ambiguity [Chambers, 1997] in which there are particular combinations of argument types for which there is no single most specific method. Within the SIR model, ambiguity is not permitted and must be handled at compile time — complex automatic ambiguity resolution mechanisms such as that found in CLOS [Paepcke, 1993] are not well–understood by developers, and type–safety requires that ambiguities cannot occur at run–time.

Since the static typing of multiple dispatch for object–oriented languages is currently at an early stage it is useful to consider the simpler approach taken by Eiffel: only one method with a given name can exist within a class (including inherited methods). This means that method selection is made only on the type of the receiver.

This approach can be used to provide a restricted version of the SIR model. However, if multiple methods with the same name and number of arguments are introduced then multiple dispatch semantics is assumed.

Method Overriding Method overriding replaces an inherited method for subclass instances. Support for method overriding is essential since it allows subclasses to polymorphically exhibit behaviour that differs from that of the superclass. The subclass version of the method will always be invoked for subclass instances, even by superclass methods that refer to it, this is the late–bound self aspect of inheritance.

Method overriding is a powerful technique but can lead to problems if the subclass modifies behaviour that is relied upon by the superclass. Within the SIR model, method overriding is only possible in contexts where the subclass inherits the realization details of the superclass. This is necessary in order for the subclass to be able to override a method without violating any constraints on the superclass. If the subclass does not have access to the full realization details of the superclass then it cannot safely override a method. Method overriding is discussed in more detail with the appropriate SIRs in this chapter.

Method Extension While method overriding is powerful it introduces a strong coupling between abstractions since, in order to safely modify behaviour, the subclass must have access to the realization details of the superclass.

Method extension describes a family of techniques that provides a restricted, but safer, form of overriding without the strong coupling associated with full overriding. The restriction guarantees that the superclass version(s) of a method will be executed, as well extensions defined in the subclass. This supports a form of inheritance in which the subclass has all of the behaviour of the superclass plus some additional behaviour. Method extension guarantees that subclass instances will exhibit all of the behaviour that superclass instances exhibit for a particular operation. The following language constructs, as described in Chapter 2 are forms of method extension:

- **before, after and around methods** — as found in CLOS and related languages.
- **inner methods** — as found in Beta.
- **call to super** — as found in Smalltalk.

The common factor across these method types is that the superclass version of the method is invoked as well as the subclass extension to the method. The constructs above support, respectively: the subclass extension being called before or after the superclass method; the subclass method being

invoked at a point determined by the superclass; and, the superclass method being invoked at a point determined by the subclass.

The SIR model does not require that direct support for all, or even any, of these method types is provided, but each will add flexibility to model description. The SIR model only refers to the general concept of method extension, not its precise implementation. Further research is required to determine the combination of extension method types that offers the greatest expressiveness with a manageable degree of complexity.

Matching The matching relationship ensures that a method can only be invoked if the types of the arguments are in a particular relationship with the type of the receiver or with each other. Most commonly, we have a binary method in which the type of the argument must be the same as that of the receiver. In subtypes, such a method will accept only subtype instances as valid arguments, for example, the equality method of a point class will only accept coloured points as arguments when inherited into a coloured point class. This behaviour cannot be supported within type systems that do not support matching (or a similar relationship), including those of mainstream languages such as C++ and Java. Matching is allowed by the SIR model since it adds power to the type system and has a sound conceptual basis — it allows specialisation of argument types as the receiver type is specialised.

4.2 Presentation of the SIR Model

The UML meta-model is used as a basis for the SIR model of inheritance. UML provides model elements to represent types and classes, it provides an association (clientship) relationship, and a foundation on which to build the SIR relationships. UML also provides a structured approach to describing new model elements such as the SIRs, the SIR semantic definitions in this chapter follow the structure of the UML semantics document [*UML Specification (draft), version 1.3 beta R7*, 1999].

4.2.1 Types and Classes

In UML, the term classifier is used to denote 'a model element that describes behavioural and structural features', the UML Class model element is the most general kind of UML Classifier and corresponds to the SIR notion of an abstraction. The UML Class model element is described as follows:

> The descriptor for a set of objects that share the same attributes, operations, methods, relationships, and behaviour. A class represents a concept within the system being modelled.

UML uses 'stereotypes' to subclassify model elements and the standard stereotypes of UML Class are `type` and `implementationClass`. A type 'is used to specify a domain of objects, together with operations applicable to the objects, without describing the physical implementation of the objects

or operations'. Implementation class is 'a stereotype for Class that provides physical implementation including attributes, associations to other classes, and methods for operations'. An unstereotyped UML class both defines a type and can provide implementation of that type.

The SIR model separates the notion of type and implementation class so the UML Class stereotypes, type and implementationClass, can be used directly in the SIR model. The UML type stereotype maps to the SIR notion of primary type. The SIR model does not however include classes that both define primary types and provide implementation, that is, instances of unstereotyped Class do not exist in the SIR model.

Note that, although we refer to implementation classes as variants and construction units in various contexts, there is no need to introduce extra model elements to represent these uses because the same implementation class could be a variant or a construction unit depending on context.

4.2.2 Representing the SIRs in the UML Meta–Model

The UML meta–model does not provide an inheritance–like relationship that is general enough to support the SIR definition but it does provide a Relationship model element that represents relationships between abstractions. The UML Relationship model element is therefore used as a basis for SIR inheritance. UML also includes a Generalisation relationship which is similar to specialisation but also encompasses variant and some uses of evolution. Due to this mismatch, SIR inheritance replaces UML Generalization in the model presented here, UML Generalization and the SIRs are not intended to be used within the same system. The UML Relationship modelling element is taken as the basis for modelling the SIR relationships as shown in Figure 4.2 (the open–ended arrows indicate generalization/specialization in the UML meta–model).

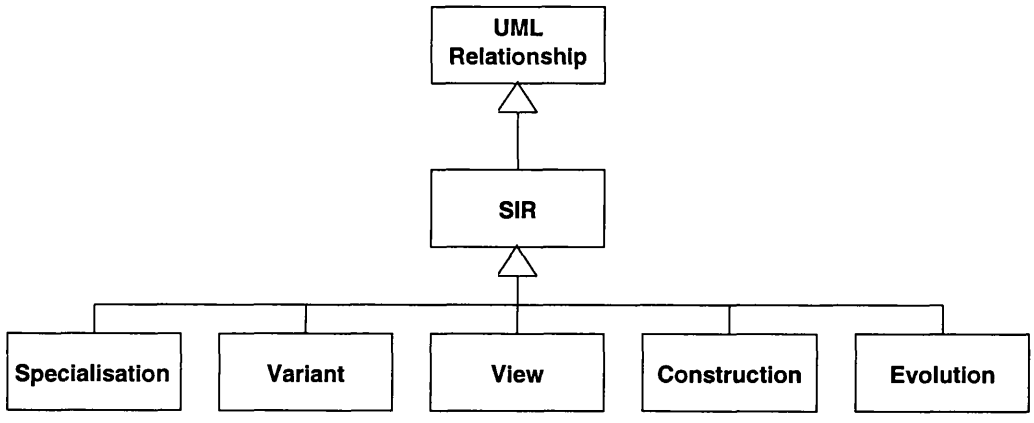

Figure 4.2: The SIR model expressed as a UML Extension.

The UML Association model element, which is defined at the class–level but introduces an instance–level relationship, is also relevant to the SIR model since the view and construction relationships are instantiated at the instance level. View and construction are therefore modelled as subclasses of both Relationship and Association in the UML meta–model.

4.2.3 Notation

The diagrammatic notation for the SIRs used in this chapter is based on the UML syntax. Rectangular boxes are used to represent Classes — the term includes both types and implementation classes in UML. For clarity we draw the boxes for implementation classes using a dashed line.

A further convention is used to distinguish between types and implementation classes, where necessary: the names of types begin with the letter 'T' and implementation classes begin with the letter 'I'. For example, TStack is a type and IArrayStack is an implementation class (a variant or construction unit depending on context).

A sans serif font is used when referring to the SIRs to indicate that the name of the SIR is being used in its specialised meaning: specialisation, variant, view, construction and evolution.

4.3 SIR

The SIR model element is a generalisation of the five specific SIRs. The semantics of SIR therefore applies to all of the SIRs.

4.3.1 Semantics of the SIR Model Element

DEFN 9 **SIR** A stated, continuing, relationship between two abstractions in which the characteristics of the first abstraction, the superclass, are also characteristics of the second abstraction, the subclass.

SIR introduces a relationship between two UML Class elements. Within the SIR model this means that the parent and child in the relationship must each be a type or an implementationClass. There is no requirement for the parent and child to both be types, or to both be classes.

Attributes

- **substitutability**
  - **yes** — if a substitutability relationship is introduced between the parent and child.
  - **no** — if no substitutability relationship is introduced between the parent and child.

The default is no and the value of the substitutability attribute is expected to be fixed by subtypes of SIR (the specialised SIR relationships).

Associations

- **superclass** — designates the Class (type or implementationClass) that is the source in the inheritance relationship.
- **subclass** — designates the Class (type or implementationClass) that is the recipient in the inheritance relationship.

Notation There is no notation directly corresponding to SIR since it is abstract and therefore has no direct instances. Subclasses of SIR (specialisation, variant, view, construction and evolution) each have corresponding graphical notations which will be introduced later in this chapter. Alternatively, the UML notation for generalisation, an open–ended arrow pointing from the subclass to the superclass, with a corresponding stereotype adornment can be used as shown in Figure 4.3, this notation is used here for clarity. In projects where the SIR model is used, the specialised notation may be preferred to avoid cluttering up diagrams with adornments.

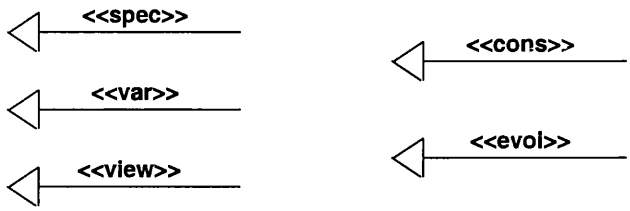

Figure 4.3: UML Stereotype notation for SIR relationships.

Standard Constraints For consistency, the standard constraints of the SIR Relationship stereotype correspond to those for UML Generalization which are reproduced here in terms of the SIR model element instead of UML Generalization:

- complete — a constraint applied to a set of SIRs with a common parent, specifying that all children have been specified (although some may be elided) and that additional children are not permitted.
- incomplete — a constraint applied to a set of SIRs with a common parent, specifying that not all children have been specified (even if some are elided) and that additional children are permitted. This is the default semantics of SIR.
- disjoint — a constraint applied to a set of SIRs with a common parent, specifying that instances may have no more that one of the given children as a type of the instance. This is the default semantics of SIR.
- overlapping — a constraint applied to a set of SIRs with a common parent, specifying that instances may have more than one of the given children as a type of the instance.

These constraints allow sets of SIRs to be characterised.

Well–Formedness Rules If the superclass in an SIR relationship is an implementation class then the subclass must also be an implementation class. In other words, it is not possible to subclass an implementation class and get a type.

4.4 SIR Specialisation

Of the five SIRs we consider specialisation first since it replaces the familiar, but confused, notion of 'is–a'.

Specialisation allows objects to be viewed at different levels with higher levels of abstraction providing less detail and therefore encompassing more objects. For example, we might have Book, Journal, Video and CD-ROM specialisations (subtypes) of a LoanItem abstraction in a library management application (Figure 4.4).

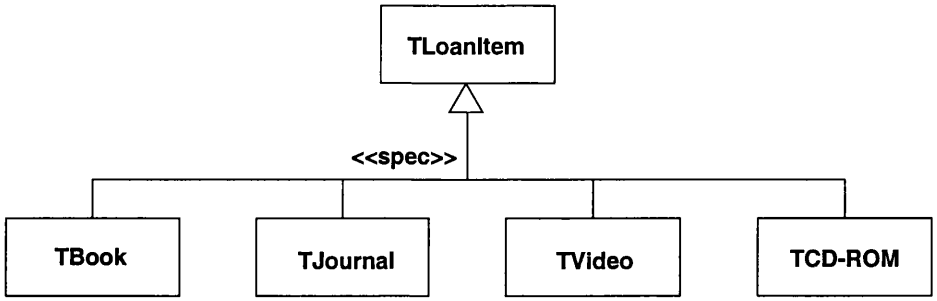

Figure 4.4: A specialisation hierarchy.

We expand upon the discussion of specialisation presented in Chapter 3 to provide a detailed semantics.

Behavioural Specialisation Specialisation leads to behavioural subtyping as well as syntactic subtyping. The SIR model does not provide a syntax for describing behavioural specifications but adopts the design–by–contract terminology of preconditions, postconditions and invariants. A behavioural specification language such as that provided by Eiffel [Meyer, 1997] could be used in conjunction with the SIR model.

The specification of a subtype must be conformant with that of its supertype(s). This means that:

- Subtype preconditions must be the same as, or weaker than, supertype preconditions.
- Subtype postconditions and invariants must be the same as, or stronger than, supertype postconditions and invariants.

These rules guarantee that a supertype client will be able to interact with subtype clients using the supertype contract. From the subtype perspective, clients may be able to meet a weaker set of preconditions and still be guaranteed the same, or stronger, results.

Should Supertypes be Abstract? It has been suggested that superclasses should be abstract in a good design [Hürsh, 1994], this rule is known as the abstract superclass rule (ASR). Since we separate the notion of types and their realizations we must ask the question 'should supertypes be abstract in a specialisation relationship?'.

The reason commonly cited for avoiding concrete superclasses is related to inheritance of implementation. Since a concrete superclass has a clear obligation to its direct clients it may make modifications to its implementation for the benefit of those clients which then have a negative effect on subclasses. The argument goes that if the superclass had been abstract then its implementor

would be considering its subclasses. (The SIR framework provides techniques for handling this situation, these techniques are described in the following chapter.) The same reasoning does not apply to the specialisation relationship since there is no implementation to inherit.

An analogous argument for an abstract *supertype* rule would be that the specification of the supertype might change to meet the needs of its clients. However, such a change would need to be performed using evolution in the SIR model, and evolution does not permit the concept represented by an abstraction to change. Either the concept should remain the same and any changes should apply unproblematically to subtypes and their clients, or the concept is different and the client must be evolved to use a different type that does represent the required concept in which case the original abstraction is left intact for its other clients. An example of the concept staying the same is the currency unit of a Money abstraction changing from the U.S. dollar to the Euro. If the default currency unit is to be changed throughout an application then all clients of the Money abstraction must be evolved. If however, a client decides that the undo functionality on a Calculator abstraction should be removed for time efficiency, it is not appropriate to remove this behaviour from the Calculator abstraction. The client must use an abstraction suited to its needs and leave the Calculator abstraction with its current functionality for other clients. The concern that a type might be changed to suit its clients is therefore not applicable within the SIR model.

A further justification for the abstract supertype rule is that concrete supertypes lead to information loss because the concrete supertype is used to represent both its direct instances and the instances of its subtypes, and there is no way for a client to determine that it is dealing with a direct supertype instance. This becomes a problem when the client needs to distinguish between objects that have a certain property (instances of a particular subtype) and instances which do not (direct instances of the supertype and its other subtypes). For example, we might have a food–processing plant application with a Bottle supertype and a BrokenBottle subtype (Figure 4.5). Clients of the bottle abstraction have no way to guarantee that a particular bottle is intact since any instance of the Bottle type may actually be a broken bottle.

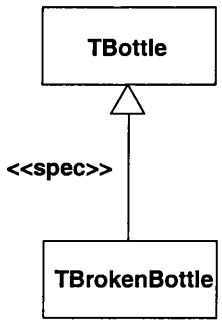

Figure 4.5: Concrete supertypes lead to information loss.

The reason that this is problematic is clear: the supertype does not describe objects that do not have the property (if it did then subtype instances with the property could not be valid supertype instances), a supertype describes objects for which the value of the property is undefined (it may be true or false) — in the example, a supertype object may be broken or unbroken, the Bottle supertype

does not say which.

Fortunately, the SIR model also provides a way of adding the additional information when it is required. We can introduce a new type which is a sibling to the type that does specify the property, the specification of this type explicitly states that its instances do not have the property in question. In the bottle example, we get a class that explicitly represents unbroken bottles (the property in question being brokenness). Clients that do not need to distinguish between objects with and without the property can use the existing supertype, clients that need to be able to identify objects without the property must use the new subtype, existing clients may be evolved to use the new subtype if required. The corresponding variants must also be evolved, ensuring that clients requesting a supertype instance will get an instance of the abstraction without the property (TBottle in the example).

In summary, this means that although failing to follow the abstract supertype rule does lead to information loss, within the SIR model that information can be introduced when it is required. It is therefore not necessary to follow an abstract subtyping rule at all times in order to benefit from its advantages.

Substitutability Specialisation is concerned with building up concepts by adding further information (specification and operations) in a subtype. The subtype must be substitutable for the supertype. But, as we saw in Chapter 2, there are multiple forms of substitutability and type systems that support them. The most common forms are subtyping and matching, and languages that support the latter typically also support the former. The SIR model permits the form of substitutability to be specified on a per Specialisation relationship basis. The default is subtyping based on the Liskov substitution principle (LSP) and it is expected that this will be the primary use of specialisation in most systems. An alternative approach would have been to introduce a separate SIR relationship to model matching. This was not considered appropriate since the conceptual relationship is the same in both cases: it is specialisation. The different forms of polymorphism that result are due to the presence of different kinds of method. That is, it is the nature of the superclass abstraction that results in a different kind of substitutability, not a difference in the inheritance relationship.

We can see the distinction between subtyping and matching forms of substitutability by considering the two possible interpretations of points and coloured points illustrated in Figure 4.6 where `SelfType` is used to indicate the current type and has matching semantics. In the first example, the methods have multiple–dispatch semantics (recall that method overloading based on the receiver type is not supported in the SIR model). This means that the subclass method will be used whenever two coloured points are compared, and the superclass method will be used otherwise (in practice we might also wish to provide methods for use when only one of the arguments is a coloured point). In the second example, the methods have matching semantics, this means that it is only possible to compare two points, or two coloured points: mixed comparisons will not type–check.

```
class Point
{
   private int x;
   private int y;

   boolean equals(Point p);
}

class ColouredPoint specialises Point
{
   private Colour col;

   boolean equals(ColouredPoint p);
}
```

```
class Point
{
   private int x;
   private int y;

   boolean equals(SelfType p);
}

class ColouredPoint specialises Point
{
   private Colour col;

   boolean equals(SelfType p);
}
```

Figure 4.6: Alternative Point abstractions with Coloured Point subclasses.

4.4.1 Semantics of SIR Specialisation

DEFN 10 **Specialisation** Specialisation allows objects to be viewed at different levels of abstraction, with higher levels providing less detail and therefore encompassing more potential objects. Specialisation is a relationship between types which leads to substitutability.

The child in a specialisation relationship is substitutable for the parent according to the rules specified by the substitutability attribute. The default is to assume UML subtyping which follows the LSP.

The specialisation relationship is transitive so if C is a subtype of B and B is a subtype of A, C is also a subtype of A.

Attributes

- **substitutability** — Always has value 'yes'.
- **substitutability details** — UML subtyping (instance-level substitutability) is the default but matching may also be specified.

Associations

- **supertype** — designates the type that is the source in the specialisation relationship.
- **subtype** — designates the type that is the recipient in the specialisation relationship.

Notation An unadorned, open–ended arrow in a UML model using the SIR extensions is assumed to be a specialisation relationship. In an environment where alternate notations for stereotypes cannot be defined then the generalisation or relationship notations may be used with an **spec** adornment.

Figure 4.7 illustrates the possible graphical notations for specialisation.

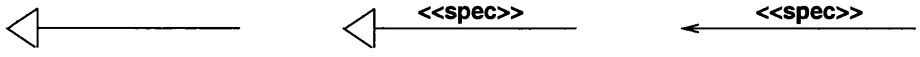

Figure 4.7: Graphical notations for specialisation.

4.5 SIR Variant

The specialisation relationship introduced in the previous section supports the creation of new abstractions by adding specification to an inherited definition, the variant relationship, on the other hand, allows realization details to be added to an inherited abstraction.

Variant supports the modelling of differences between sets of objects that do not cause them to be classified differently. For example, we might have dense and sparse implementations of a matrix, as shown in Figure 4.8.

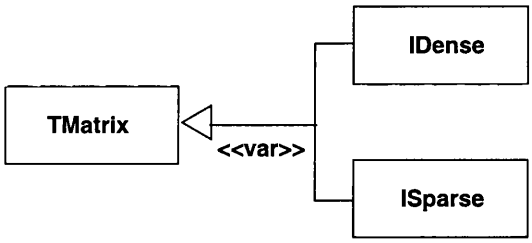

Figure 4.8: Dense and Sparse variants of Matrix.

No Separate Implements Relationship In most programming languages and modelling languages where type and realization can be, or must be, separate, the relationship between a type and a realization of that type is not represented in the same way as the relationship between a partial realization and a more complete realization. The nature of the relationship is the same in both cases: realization is added indicating that some instances of the inherited abstraction will have the described internal behaviour. When the inherited abstraction is a type, it can be considered to be a trivial realization, that is, it specifies no realization detail.

Both UML and Java have separate mechanisms for relating a type and an implementation of it (`realization` for UML and `implements` for Java) and for relating a partial implementation with a more complete implementation (`generalization` for UML and `extends` for Java). In Java the same construct, `extends`, is used to relate two types (interfaces), and to relate two implementations (the situation in UML is analogous). The SIR model uses specialisation to introduce further specification (including operations), and variant to introduce realization. This means that the superclass in a variant relationship may be a type or an implementation class, in both cases, the specification and realization of the superclass are inherited. When the superclass is a type, there is no realization to inherit so only type information is inherited.

The distinction between specialisation and variant is made in the SIR model because the two relationships are conceptually orthogonal: with variant we are realizing existing (partially realized) behavioural specifications, and with specialisation we are extending behavioural specifications.

Specialisation and Variant It is important to understand the way that specialisation and variant interoperate. It is often the case that the variant hierarchy will shadow the specialisation hierarchy,

a subtype variant is likely to want to specialise and/or extend the realization provided by a corresponding supertype variant. Conformant overriding allows inherited methods to be specialised and a specialised version of an inherited method should be behaviourally compatible with the supertype version, this means that the only allowable modifications are to provide the same behaviour in the context of the information available in the subclass.

An example of the variant hierarchy shadowing the specialisation hierarchy occurs when we have a specialisation hierarchy of LoanItems in a library application (with subtypes Book, Journal, and so on), shadowed by corresponding variant implementations. TLoanItem is an abstract type, and it has a corresponding abstract implementation, ILoanItem, which provides implementation that is shared by all loanable items, as shown in Figure 4.9. In order to implement the concrete subtypes of TLoanItem, we introduce IBook and IJournal which inherit their basic behaviour from ILoanItem. This is shown in Figure 4.10.

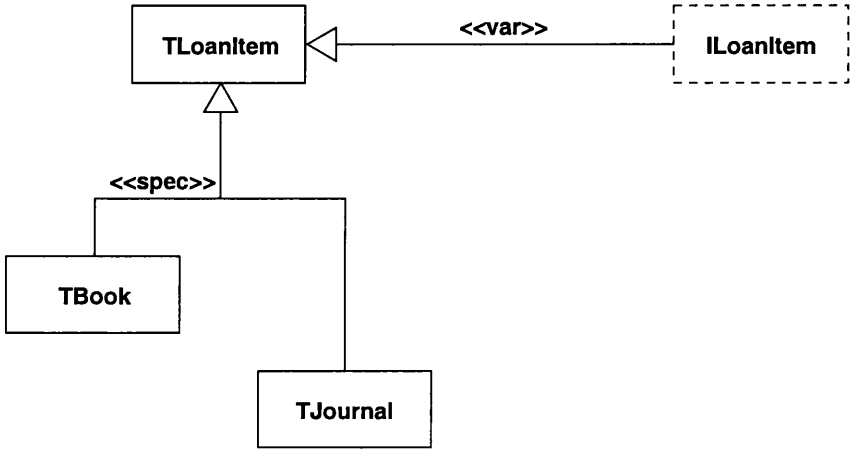

Figure 4.9: An abstract primary type with corresponding abstract implementation.

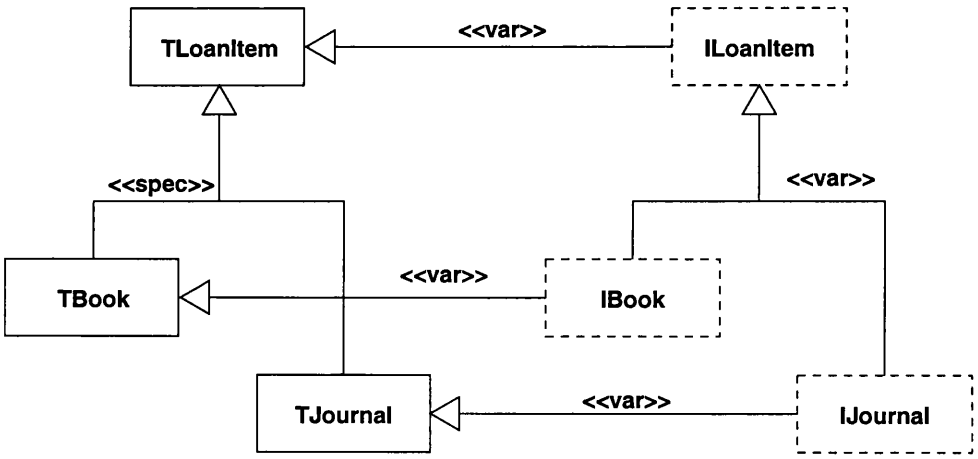

Figure 4.10: Variant hierarchy shadowing specialisation hierarchy.

Note that the same structure could also result from a concrete supertype and its subtypes with corresponding concrete variants.

Behavioural Specification As with specialisation, variant also follows the rules of design–by–contract. Any specification introduced in a variant relationship will be concerned with the implementation of the abstraction rather that its external behaviour. The specification of a variant must be at least as strong as that of its superclass in order to maintain the inherited contract.

The contract of a subvariant may be stronger than the contract of its supervariant in two ways: (i) a subvariant may introduce stronger implementation invariants in the subvariant and then the subvariant implementation must meet the same contract in the context of the subvariant, or (ii) the subvariant may have a primary type which is a subtype of the primary type of the supervariant, in which case the subvariant may have a stronger contract to meet.

A subvariant may inherit methods with a weaker contract than that in the subvariant, the subvariant must then meet its stronger contract by overriding the inherited method. This may mean that, for some methods, additional actions need to be performed and in this case method extension should be used in preference to full overriding, if possible. The IBook variant in 4.10 is an example of a subvariant which has a stronger contract to meet than the realization it inherits. IBook inherits the realization of ILoanItem which meets the weaker TLoanItem contract, IBook must specialise the inherited realization to meet the stronger TBook contract.

Method extension is a useful technique for extending a supervariant method to meet a stronger contract in a subvariant. The supervariant method can be used to meet the weaker contract while the method extension does the additional work required to meet the subvariant contract. Method extensions must obey the following rules:

- A method extension executed before the inherited method must establish the precondition of that method in the superclass.
- A method extension executed after the inherited method must maintain the postcondition of that method in the superclass.

These rules are in addition to establishing the postcondition of the method extension given its precondition.

No Need for Public/Protected Distinction A useful secondary outcome of the SIR model of inheritance is that the usual distinction between public and protected methods is not required. Methods introduced in a primary type are automatically public, while those associated with a variant are not publicly visible — this enables clients to access all variants through the same primary type. A variant may introduce new methods but they will not be accessible from primary types; they will only be visible via the view, variant or construction relationships.

4.5.1 Semantics of SIR Variant

DEFN 11 **Variant** An inheritance relationship which supports the modelling of differences between sets of objects that do not cause them to be classified differently.

The variant relationship does not introduce a primary type so the subclass must always be an implementation class. Variant trivially leads to subtyping since the subclass has the same type and behavioural specification as the parent. Variant allows operations to be fully or partially realized and allows existing methods to be extended or overridden, the specification of implementation and actual implementation may be introduced, and constructors for instances of the primary type may be introduced.

The variant relationship is transitive, so if C is a variant of B and B is a variant of A, C is also a variant of A.

Additionally, a variant of a subtype can also be said to be a variant of its supertype.

Associations

- **supervariant** — designates the type or implementationClass that is the source in the variant relationship.
- **subvariant** — designates the implementationClass that is the recipient in the variant relationship.

Notation Variant is depicted using a dashed, open–ended arrow. (This is the same notation as UML realization, but UML realization is not required in SIR designs so there should be no confusion.) As with the other SIR relationships, a **var** adornment may be used if alternative notation cannot be specified. The graphical representations of the variant relationship are shown in Figure 4.11.

Figure 4.11: Graphical notation for variant.

4.6 SIR Construction

Whereas variant introduces a (partial) realization of an abstraction, construction allows existing realization to be reused by a new abstraction. For example, a stereo system abstraction could be constructed from amplifier, radio, CD player and tape deck abstractions as shown in Figure 4.12. Methods may be renamed, if required, in a construction relationship, this may be necessary to avoid ambiguities, or simply because another name is more appropriate in the new context.

We now discuss a number of issues concerning construction to provide a detailed semantics of the relationship.

Construction and Forwarding Construction does not require forwarding of operation invocations from the subclass instance to the superclass instance. The superclass methods are inherited into the subclass, this means that superclass methods can directly implement subclass operations without

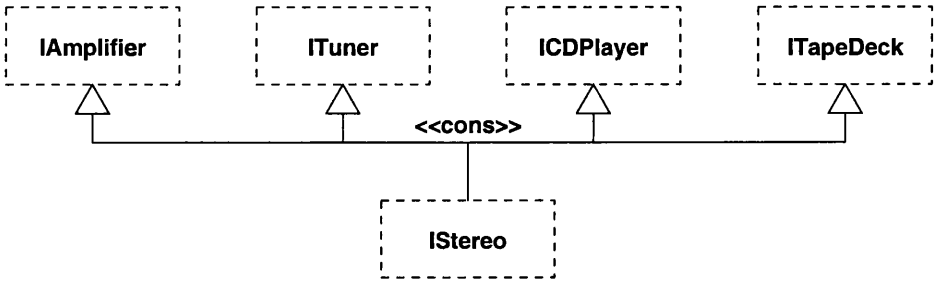

Figure 4.12: Construction of a stereo abstraction from its constituent parts.

explicit forwarding. For example, a subclass which is a variant of a primary type will inherit its operations from that primary type, if the subclass is constructed from a superclass which provides methods corresponding to the inherited operations then those methods will be used when a method is invoked on the subclass.

Construction, Overriding and Sharing Construction is the SIR that directly supports code reuse. It does not support overriding but it does operate at the instance level, allowing superclass instances to be varied dynamically and shared by multiple subclasses. For example, we might have multiple copies of the same book in a library (for example, Gulliver's Travels by Jonathan Swift) and by constructing a Copy subclass from a Book superclass (Figure 4.13) each copy can be constructed from the same instance of Book so that each subclass instance shares the same superclass instance.

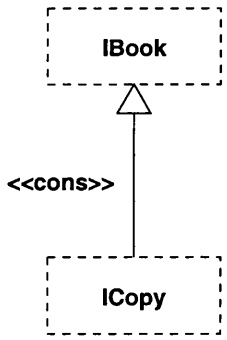

Figure 4.13: The construction relationship between Book and Copy supports multiple copies of the same work.

Overriding is not supported by construction because there is a trade–off between sharing supertype instances and permitting overriding. For example, suppose we were permitted to construct a Square abstraction from a Rectangle abstraction by introducing the constraint that the dimensions are equal, and overriding methods to support this constraint. Now suppose we construct a Window abstraction from Rectangle (since the behaviour of windows includes that of rectangles). Since sharing of superclass instances is permitted we could have a Window instance and a Square instance sharing the same Rectangle superclass instance. The problem occurs because the Window does not know about the constraints imposed on the superclass by the Square abstraction and may therefore violate those constraints by changing the aspect ratio of the Window. Disallowing overriding within construction

removes this problem since it prevents the Square abstraction from introducing a constraint on Rectangles.

In most cases, the lack of overriding within construction is not a problem because another SIR — variant — offers overriding but not sharing. A technique for combining the variant and construction relationships to achieve 'programming–by–difference' is described in the following chapter. This technique helps if the specialised Square behaviour can be modelled as a valid variant of Rectangle. Under some interpretations of the relationship between Square and Rectangle, this is not possible, this problem is discussed in detail in Chapter 6 where the square/rectangle problem is analysed in terms of the SIR model.

Removing overriding from a code reuse relationship by providing it elsewhere in the SIR model means that construction can support shared supertype instances in a safe manner.

Construction from Types and Classes The subclass in a construction relationship is always an implementation class but the superclass can be a type or an implementationClass. In either case, the superclass instance is accessed via its type (primary or secondary, respectively). Construction based on an implementationClass indicates that a particular variant is required for the construction relationship.

For example, a Window abstraction may be constructed from a Scrollbar type so that an appropriate variant can be used depending on the current system preferences such as the width of the scrollbar, and whether it should appear on the left or the right of the window (Figure 4.14). On the other hand, an abstraction representing a map of grid locations might be constructed specifically from a sparse variant of the matrix abstraction for space efficiency (Figure 4.15). It is valid to refer to objects via a secondary type rather than a primary type in an implementation class although it should be recognised that this increases the coupling between the abstractions.

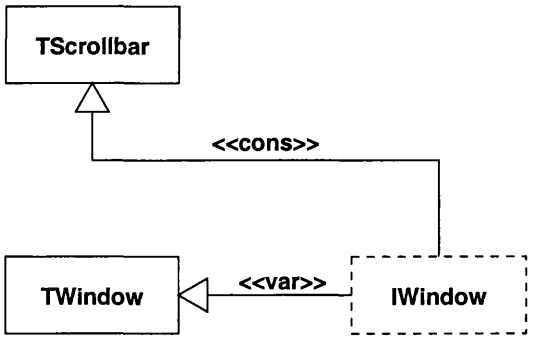

Figure 4.14: Construction from a primary type.

Construction versus Manual Forwarding Since construction does not support overriding it is appropriate to consider whether construction could be replaced by basic clientship, or by association with manual forwarding. In fact, since construction introduces an object–level relationship it is a kind of association relationship (represented by Construction being a subclass of UML Association) and it is therefore meaningless to discuss construction versus association since construction is a form

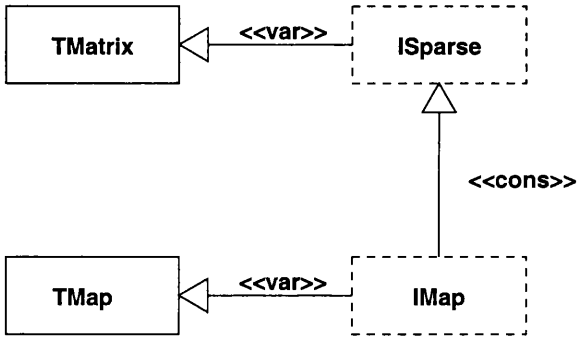

Figure 4.15: Construction from a primary type.

of association. That being the case, we must address the issue of whether it is necessary to identify construction as a distinct form of association.

One reason to model construction explicitly is that there is a conceptual distinction between construction and other forms of association. The behaviour acquired via a construction conceptually belongs to the inheriting abstraction: the characteristics of the superclass are also characteristics of the subclass, as required by SIR inheritance. Using construction emphasises this. For example, there is a distinction between the relationship between a window and a graphical rectangle abstraction (where the behaviour of the rectangle is also the behaviour of the window — moving or resizing the rectangle has the same effect on the window), and the relationship between a drawing application window and the graphical objects within it (turning a contained circle blue does not make the window blue). In the former case, construction is appropriate whereas in the latter clientship should be used.

A further reason for separating construction from other forms of clientship is that construction implies automatic forwarding so that wrapper methods, to explicitly forward methods from client to server, need not be created. If the inherited interface changes (for example a new method is added) then there is no need to change the subclass, the changes will be propagated automatically.

Construction and Delegation Delegation is a powerful inheritance–related technique which is found in object–based languages. Delegation allows objects to be built from similar objects simply by specifying the differences. For example, if we have a blue circle object, it is possible to create a red circle by specifying the colour and then delegating all other properties to the blue circle object. True delegation supports overriding so that an object may behave differently depending on its caller. This means that when the draw operation in the blue circle is invoked via the red circle, a red line will be drawn rather than a blue one.

Construction does not have full delegation semantics. To model differences in data element values, as in the circle example, different instances of the same class would be made, perhaps by cloning the blue circle and then changing the colour of the copy. To model differences in behaviour, the variant relationship should be used. This is appropriate in a class–based language where abstractions are modelled at the class level: each variation in behaviour should be modelled via a distinct class.

A further form of delegation, in which method extension can be supported on a per client basis, is handled by the view relationship (see Section 4.7). The only form of delegation that is not handled within the SIR model occurs when full overriding is required, coupled with a continuing relationship with a superclass object. This form of delegation is not safe — it is the disallowed construction relationship between square and rectangle from Section 4.6. This problem is recognised in object–based languages supporting delegation but safety is a lesser concern than flexibility in such languages [Chambers et al., 1991].

Construction and Implementation Hiding We have already noted that there is no need to distinguish between public and protected methods in the SIR model. We now explain why there is no need to distinguish between protected and private methods either.

If a superclass wishes to hide implementation details even from subclasses then it can introduce a secondary type defining the methods that should be visible from subclasses and forward to that abstraction, via construction. The private part of the implementation is then placed in a subvariant of the secondary type with the operations that require direct access to it. Only the operations accessible via the supervariant type will be visible to subclasses (and other methods in the superclass).

This approach means that there is no need to distinguish between public, private and protected methods within the SIR model, providing a significant reduction in complexity.

4.6.1 Semantics of SIR Construction

DEFN 12 **Construction** Construction allows existing implementation to be reused in the realization of a new abstraction.

Construction can be modelled as a subtype of UML Association since it introduces an instance level relationship between superclass and subclass objects. Construction does not lead to a subtyping relationship.

A construction relationship between superclass and subclass means that a subclass instance relies on a superclass instance for some or all of its behaviour. If the superclass offers methods that directly implement some of the subclass operations then they are made directly available to clients; if the subclass operations can be implemented in terms of the superclass methods then the parent's methods are available only within the subclass.

Multiple subclass instances may rely on the same superclass instance. Superclass instances may be changed dynamically. A subclass instance may also be constructed from multiple superclass instances.

Construction is transitive so if C is constructed from B, and B is constructed from A, then we can say that C is constructed from A.

Attributes

- **multiplicity** — the number of superclass objects from which the subclass can be constructed, usually one or many, but may take other values as required. (See the discussion on Multiple Inheritance in Section 4.9.)

Associations

- **superclass** — designates the type or implementation class that is the source in the construction relationship.
- **subclass** — designates the implementation class that is the recipient in the construction relationship.

Notation Construction is depicted using a filled arrow. As with the other SIRs, a **cons** adornment may be used if alternative notation cannot be specified. The graphical representations of construction are shown in Figure 4.16.

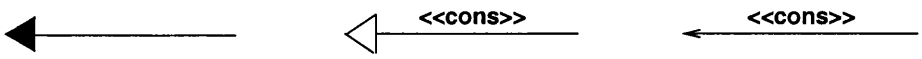

Figure 4.16: Graphical notation for SIR Construction.

4.7 SIR View

View supports client–specific behavioural variation for abstractions. For example, we can have staff and student views of a postgraduate student who is also a teaching assistant.

View as a Relationship between Type and Implementation Class At first it may seem that view should be a relationship between two types, after all it supports instances of one type behaving as if they were instances of another type. This approach was considered but would require adopting one of two, unsatisfactory, solutions:

1. Introducing a dependency from the view type to the target which is only valid for a subset of all instances. For example, if we introduced a relationship from a Point view type to a Pair target type (enabling pairs to be viewed as points) then we would introduce a dependency from Point to Pair which only applies to a subset of instances of Point — those implemented as a view of a Pair.
2. Treating view types as special types and introducing a new primary type for every view–target combination. The new primary type would need to be a specialisation of the subtype in the view relationship. This would overly complicate the model and the introduced primary types would not have a conceptual rôle — they would not represent distinct domain–level abstractions.

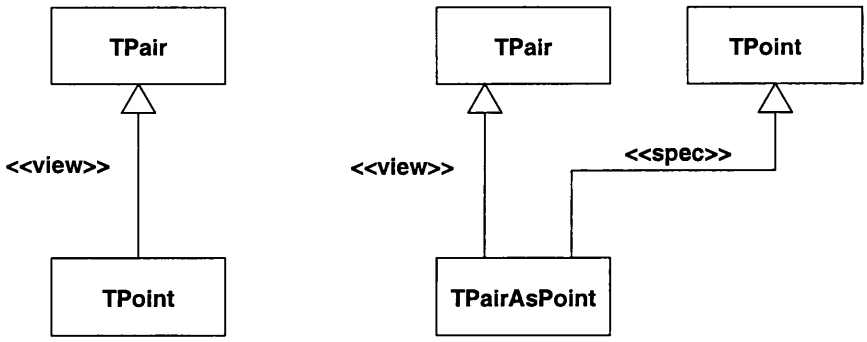

Figure 4.17: Rejected approaches to supporting the view relationship.

These rejected alternatives are illustrated in Figure 4.17.

Within the SIR model, behaviour appropriate to a subset of all instances of a type is handled using the variant relationship and this also applies in the case where a particular subset of the instances of a type form a view of another type. Therefore view is naturally modelled as a relationship between a variant of the view subtype, and the target type. The view type is an ordinary, possibly pre–existing, type which may have other variants with no dependency on the target type. Figure 4.18 shows the chosen approach where the IPairAsPoint class is a view of TPair which is also a variant of TPoint, allowing a pair to be viewed as a point.

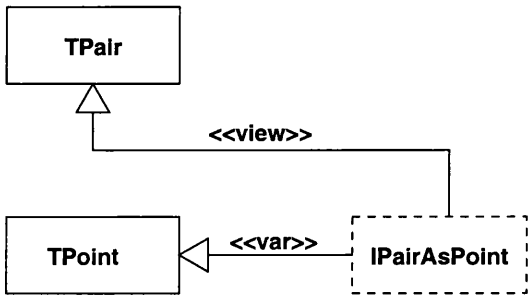

Figure 4.18: View in the SIR model.

View is used in combination with variant to create a variant of one type that provides a view of another. In the SIR model a type is not considered special simply because it can be realized in terms of another type. In order to show that one type can be viewed via second type, a variant of the second type is introduced which is a view of the first.

Note that the realization information introduced by such a variant is that its implementation will be in terms of the target type. Only implementation that applies to all view instances should be introduced within such a variant. This allows multiple variant implementations of the same view. A type that has variants that are views of another type may also have other variants which are not based on the target. There may be variants which provide a view of another type and/or direct variants. For example the TPoint type in Figure 4.18 may have a variant IPoint which implements points directly, instances of both variants can be used interchangeably via the TPoint type.

Views of Types and Classes So far, we have considered the superclass in a view relationship to be a type. We should also consider creating views of implementation classes: this is allowable within the realization domain, but if we have clients that access such target objects via their primary type then it would not be possible for them to create an instance of such a view. If a client, via a primary type, is to be able to create a view instance of a target object then the view must be specified on the target type and must apply to all target objects. It also follows that a subvariant must be viewable as a certain type if any of its supervariants (including a primary type) are viewable in terms of that type.

The main use of view is expected to be in allowing instances of one primary type to be viewed as instances of another. Such views must be specified in terms of their target type — this is necessary since for type safety it must be possible to implement a view of any variant of a type, dependence on the details of a particular variant would preclude this. For many views the type should provide sufficient information for an elegant and efficient implementation. This means that a single view class can apply to instances of any variant of the target type.

In other cases, lack of publicly available information or the need for efficiency may mean that a view needs to be modelled with respect to a particular implementation of the target. Such a view implementation must specialise the view relationship so that it is a view of a particular variant of the target — in such cases sufficient views must be created to handle all variants of the target. This may seem undesirable but views are conceptually tightly coupled to their targets. Additionally, in traditional models all views would be implemented within the target (the target would inherit from types corresponding to each view and a single abstraction offering all views would result).

In order to maintain type safety, it must be possible to view all instances of a target via a particular view. This means that if views are implemented in terms of implementation classes then all implementation classes must have a corresponding view. This need not be a 1-1 correspondence: the view corresponding to a supervariant can also apply to its subvariants.

Views and Method Overriding/Extension A view is not permitted to alter the behaviour of its target to other clients of the target but it should be able to modify the behaviour of an object while it is being viewed in a particular way. Methods on the view must maintain any constraints on the target so that the target is a valid target instance after a method invocation on the view. In order to achieve this with full overriding, the view would need to know about any constraints introduced by the implementation so that it could maintain them. For views that depend only on the type of the target, full overriding would not be safe: a view cannot override a method if it does not understand the full behaviour of that method and it cannot understand the full behaviour because the constraints introduced by the implementation are not described within the specification of the type.

Although full method overriding is not safe, we can permit a limited form of method extension. A method extension that does not modify the target object can safely be added in a view. Such method extensions can modify the state within the view and can be used to perform actions specific

to the view. Method extensions will be executed in addition to the corresponding target method when invoked by a target method that is executing within a view.

At first it seems that full overriding could be permitted for views that are based on a particular variant rather than a primary type and therefore have access to realization details. Further examination shows that this is not the case. The view may be attached to a subclass of the variant at run-time. This subclass may introduce additional constraints of which the view is not aware. Full overriding is not safe in this case either.

Methods can be extended in a view relationship but the extensions will only be invoked through the view; other clients get the original behaviour. This is illustrated in Figure 4.19, a printer abstraction which handles the printing of documents is viewed as a printer abstraction that outputs a banner page (containing administrative information about the document being printed) before the document is printed. When a view object is instantiated the target object retains its original behaviour, only clients accessing the target via the view object will see the extended behaviour.

```
type TPrinter
{
   print(); // print the current document
}

class IPrinter
      variantOf TPrinter
{
   print()
   {
      // print current document
   }
}

type TPrinterWithBanner
{
   print(); // print a banner page and then the current document
}

class IPrinterWithBanner
      variantOf TPrinterWithBanner
      view TPrinter
{
   before print()
   {
      // print banner page
   }
}

TPrinter p = new IPrinter();
TPrinterWithBanner bp = view p as IPrinterWithBanner();

p.print(); // prints current document without banner page
bp.print(); // prints banner page then current document
```

Figure 4.19: Method extension in view.

This use of view provides a safe form of object–level delegation with an appropriate conceptual relationship. The view relationship supports method extension rather than full overriding, since, as discussed above, safe overriding is not possible for view relationships. The same is true for delegation in object–based languages such as Self [Chambers et al., 1991], but there, flexibility is considered more important than safety. The SIR approach to delegation is a useful compromise

between modelling power and safety.

View and Identity The identity of a view instance is composite. It is based on the identity of the target and it should be possible to determine whether two objects are views (including the default view) of the same object. For example, it should be possible to determine whether a given student and member of staff are actually the same postgraduate teaching assistant. It should also be possible to distinguish between different view instances with the same target instance: we might have two instances of the same view, or, an instance of each of two views which have the same target object. For example, multiple departments in an organisation might have views of the same customer and billing of the customer should not be considered complete just because billing has been carried out with respect to one of the views.

These two notions of identity should be sufficient to handle situations where views should be considered the same and situations where they should be differentiated. A similar duality of identity is found in subclasses where there is also a superclass identity, and in the inner class construct of Java where the inner class has its own identity which is based on the identity of the outer class. The identity of a view object is an extension of the identity of the target.

State–Based Views It is sometimes the case that a particular view is only valid when its target is in a certain state. For example, a credit card may only be a valid payment method if it has not expired. It is straightforward to only allow the creation of view objects when the target is in an appropriate state so the problem is how to prevent the view object being used when the target object leaves the defining state — for example, the credit card expires.

In such cases, the target being in an appropriate state is an invariant of the view. The target moving outside that state will make the view invalid. An operation must be made available so that clients can check that the target is in the required state *before* invoking any methods on the view. This is the design–by–contract approach. An alternative would be to use the exception mechanism of a language. The client would then have to deal with the exceptional case that the target object is not in the required state. Either approach can be used within the SIR model.

Bi–Directional Views As the subtypes in view relationships are indistinguishable from other types in the SIR framework it is possible to build a system in which two types can be viewed in terms of each other. For example, we might view a 2–D Point as a Pair and vice–versa as shown in Figure 4.21. Using view, we create variants of both point and pair types which are also views of the other type, therefore allowing instances of each type to be viewed as instances of the other.

```
TPair pair1 = new IPair;
TPoint point1 = view pair1 via IPairAsPoint;
TPair pair2 = view point1 via IPointAsPair;
```

Figure 4.20: Bi–directional view.

If we start by instantiating a Pair and then view it as a Point, we can then view the resulting Point

as a Pair, this situation is shown in pseudo code in Figure 4.20. In most scenarios it is likely that the result should be the original Pair, rather than a new instance which is a view of a Point which is a view of a Pair. This is the default behaviour in such cases in the SIR model.

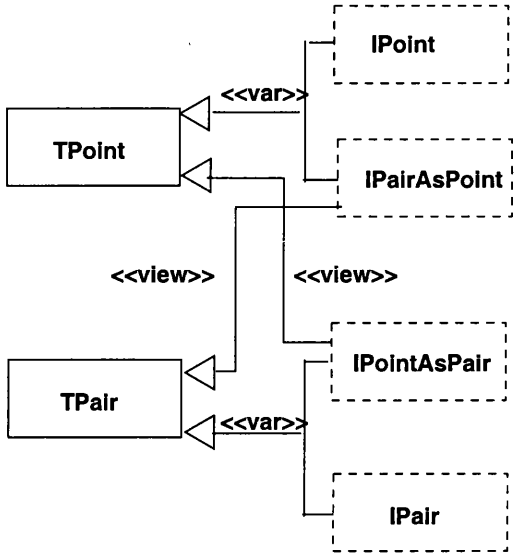

Figure 4.21: Bidirectional view.

When implementing the second of the two views it becomes necessary to provide behaviour in IPairAsPoint and IPointAsPair (in Figure 4.21) to return the target rather creating a new instance.

4.7.1 Semantics of SIR View

DEFN 13 **View** View supports client–specific behavioural variation for abstractions.

The instances of a subclass in a view relationship get their underlying behaviour and identity from an instance of the superclass (usually a type). View allows a supertype instance to be viewed as if it were an instance of the type which the subclass implements.

Attributes

- **substitutability** — 'no': view never leads to substitutability.

Associations

- **target** — designates the type or implementationClass that is the source in the view relationship.
- **view** — designates the implementationClass that is the recipient in the view relationship.

Notation View is depicted using a double–headed arrow. As with the other SIR relationships, a **view** adornment may be used if alternative notation cannot be specified. The graphical representations of view are shown in Figure 4.22.

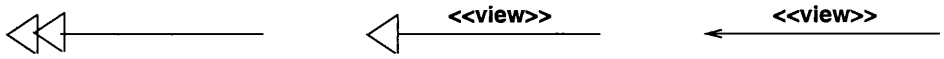

Figure 4.22: Graphical notation for view.

4.8 SIR Evolution

The last of the five SIRs in evolution which supports the adaptation of an existing abstraction in order to meet new or changing requirements. For example, evolving a taxation abstraction to meet new European laws, as shown in Figure 4.23.

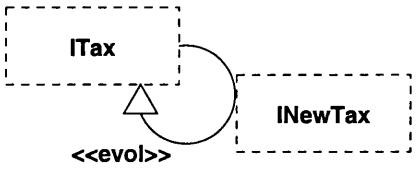

Figure 4.23: Evolution of a taxation abstraction to meet new laws.

Evolution as Refinement Evolution can be seen as a form of refinement. Refinement (as in Morgan's Refinement Calculus [Morgan, 1994]) moves an incomplete implementation towards a complete implementation. Unlike variant, evolution does not model the (partial) realization of a type for a subset of all instances. Instead, evolution is the refinement of an abstraction (which may be a type or implementation class) towards its ideal representation in a model.

Evolution occurs when information about an abstraction which was not included when the abstraction was first conceived must now be incorporated into the abstraction. There are a number of reasons why information may not have been incorporated into an earlier version of the abstraction. It may not have been available or even in existence, it may have been deliberately ignored for simplicity or it may not have been considered relevant to the system under development. Even with perfect foresight, evolution would be a useful relationship. It allows abstractions to be built up layer by layer, reducing the cognitive overhead in understanding and maintaining those abstractions and clearly separating parts of the implementation that are not interdependent.

Evolution versus View In some situations it may seem that view would be a valid alternative to evolution. Rather than changing the original abstraction to suit new purposes a filter is built so that instances of the original abstraction can behave differently in a new system. For example, if we decide that it should be possible to iconize all graphical window objects, we could add iconize behaviour in a view and access windows via that view so that they can be iconized. Care should be taken not to confuse view and evolution. They have very different semantics. Using evolution will change the behaviour of an abstraction *throughout the system* including instances that are generated by other reused components. View will only change the view of objects when they are viewed in a certain way. In the example, windows could be created by parts of a system that do not use the

view and such windows would not be iconizable, since the requirement was that all windows should be iconizable, the view relationship was not appropriate in this case. Both relationships are valuable in different circumstances and model different conceptual relationships.

State–Based Evolution It is a common mistake for software developers to try and model the behaviour of the same abstraction in different states using subclasses of a base abstraction. This fails because it is not possible for an object represented in this way to change state without losing its identity (that is, a new object must be created which is in the new state).

It is tempting to try and model different states using variant, after all variants model a subset of the instances of a class. However, variants do not model a subset of the state–space of an abstraction, this is different to modelling a subset of instances throughout their lifecycle. The definition of the variant relationship could be extended to allow objects to dynamically move between variants but this would go against the maxim of keeping each SIR relationship as simple as possible.

In fact, there is no need to complicate the variant relationship in this manner since evolution is the correct relationship for modelling state–specific behaviour. In adding a set of state–specific behaviours (specialised specifications and/or implementations) to an abstraction we add detail to the definition of an existing abstraction — this is exactly what evolution is intended to do. Evolution can be used to define each set of state–specific behaviours independently and to combine them into a complete definition of the abstraction. Shared behaviours are defined on the base abstraction. An example of state–based evolution is the modelling of a List abstraction in terms of its Empty and NonEmpty states, behaviour which is shared across states is implemented in a List abstraction and then evolution is used to add Empty and NonEmpty adaptations which implement state–specific behaviour, as shown in Figure 4.24. This approach is described in more detail in the following chapter.

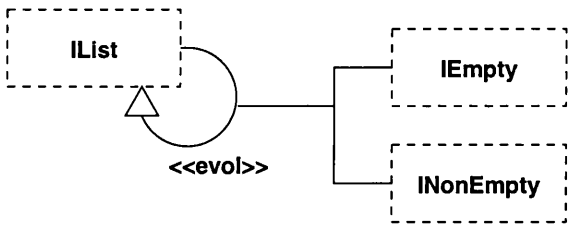

Figure 4.24: State–based evolution.

Dependencies between Adaptations Some adaptations of an abstraction may depend on others. If an adaptation that is depended upon is not required (that is, some systems may choose not to adopt it) then the depending adaptation must specify the adaptation(s) that are depended upon as parents in an evolution relationship. For example, a Dollars adaptation of a Money abstraction that uses Sterling as its currency, may require that the currency–conversion adaptation has been applied (see Figure 4.25). Adaptations that do not have specified dependencies may be applied in any order.

A software development environment should provide a mechanism for registering adaptations so that

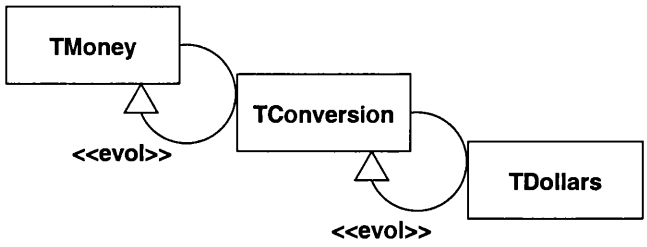

Figure 4.25: There may be dependencies between adaptations.

any ambiguities can be highlighted and resolved. The newest adaptation will be required to resolve any ambiguities before it can be registered as an adaptation of the original abstraction.

Evolution and the Open/Closed Principle Inheritance is often given as an example of a relationship that supports the Open/Closed principle: new behaviour can be introduced based on existing behaviour *but without modifying existing code* or violating contracts with existing clients. Evolution supports the modification of existing abstractions so it may appear to violate the Open/Closed principle. However, evolution does not violate the Open/Closed principle since modifications are described separately from the original abstraction with the original abstraction remaining intact, since evolution is conformant, contracts with existing clients are also maintained. Evolution therefore adds expressive power to object–orientation without violating the Open/Closed principle.

4.8.1 Semantics of SIR Evolution

DEFN 14 **Evolution** Evolution supports the adaptation of an existing abstraction in order to meet new or changing requirements.

The superclass and subclass in an evolution relationship represent the same concept at different points in its development.

Attributes

- **status** — must have the value **required** or **optional**. All systems that use the original abstraction must also use a required adaptation. Optional adaptations must be explicitly requested by a system.
- **substitutability** — 'yes'.
- **substitutability details** — the subclass in an evolution relationship must be class–substitutable for the superclass. Subtyping substitutability is not required since supertype and subtype instances cannot coexist in any system.

Associations

- **existing abstraction** — designates the type or implementationClass before the evolution relationship is applied.
- **resulting abstraction** — designates the type or implementationClass after the evolution relationship is applied.
- **adaptation** — designates the abstraction describing the differences between the existing abstraction and the resulting abstraction.

Notation Evolution is depicted using an arrow from an abstraction and back to itself, the adaptation is placed on the arrow as shown in Figure 4.26. The notation indicates that the superclass and subclass in the relationship are the same abstraction, the adaptation or adaptations that are being applied are attached to the arrow. As with the other SIR relationships, an **evol** adornment may be used if alternative notation cannot be specified. The graphical representations of evolution are shown in Figure 4.26.

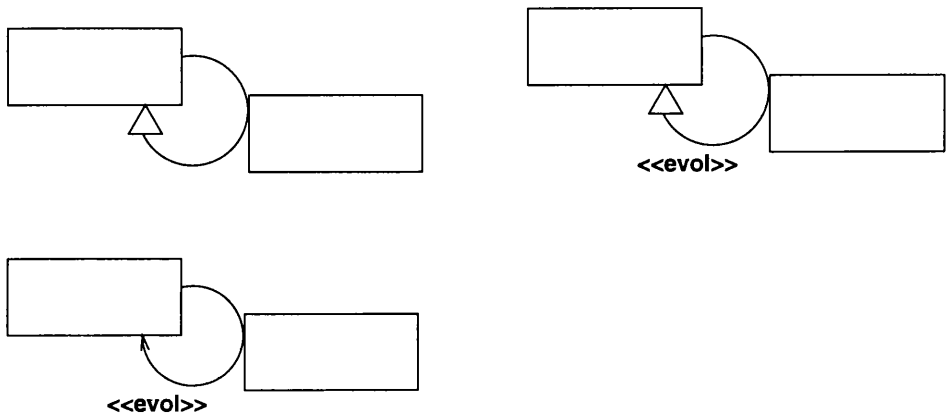

Figure 4.26: Graphical notation for evolution.

Well–Formedness Rules The superclass and subclass in an evolution relationship must either be both types or both implementationClasses. Evolution cannot move an abstraction between Class stereotypes.

4.9 Multiple Inheritance in the SIR Model

The SIR model provides a new framework in which to understand multiple inheritance. In the past it has only been possible to contrast multiple interface inheritance with multiple implementation inheritance, and the combination of interface inheritance and implementation inheritance often referred to as a 'marriage of convenience'. It is now possible to analyse multiple inheritance in greater detail.

4.9.1 Homogeneous Multiple Inheritance

Homogeneous multiple inheritance refers to the situation where a subclass has multiple superclasses via the same SIR relationship. For example, a decorative wrist–watch might be classified as both a timepiece and as an item of jewellery, warranting multiple specialisation (see Figure 4.27).

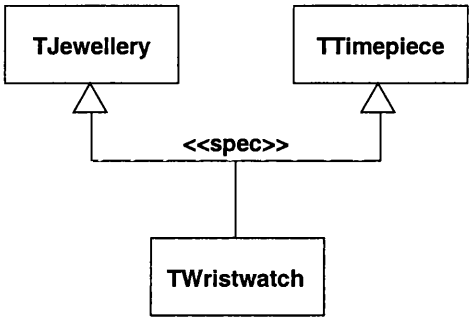

Figure 4.27: Multiple specialisation.

Multiple Specialisation Multiple specialisation can be used to model concepts that have multiple valid classifications, each of which is useful within the system and each of which contributes to the inheriting concept. This is the case with the wrist–watch example of multiple specialisation in Figure 4.27.

Multiple specialisation is useful but it is expected that it will be used less in systems designed using the SIR model than in other models. This is because multiple specialisation is often a side effect of other relationships which are modelled using different SIRs in the SIR model. Since specialisation does not occur as a side–effect of other relationships within the SIR model, multiple specialisation is restricted to the case where a subtype inherits its intrinsic characteristics from two or more supertypes.

Multiple Variant Multiple variant may be used in situations where multiple realization aspects vary independently. For example, we may have variants of a Window abstraction with and without scrollbars and, with and without titles. These aspects can be captured in variants with the variant realizing windows with scrollbars and titles inheriting from two supervariants.

All of the supervariants in a multiple variant relationship must have a common supervariant. This includes the case of variants with primary types that are in a specialisation relationships since a variant of a subtype can also be considered to be a variant of its supertypes. Figure 4.28 provides an example of a valid multiple variant relationship. Within a backup system we have a TMedia supertype corresponding to the various types of media that may be used, this type is abstract and has a corresponding variant implementation; variants of subtypes of TMedia, such as TCDROM, can inherit the IMedia realization through a variant relationship. This is permitted since TMedia and TCDROM are in a specialisation relationship, and the two supervariants of ICDROM, TCDROM and IMedia, are considered to have a common supervariant, TMedia.

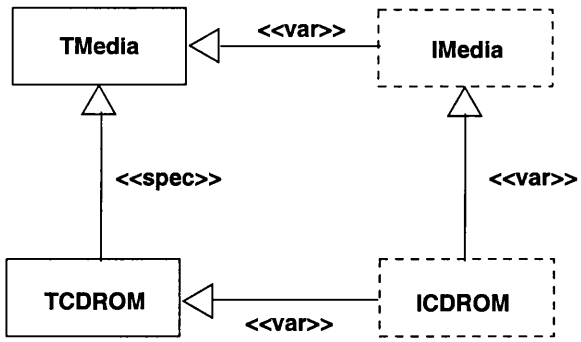

Figure 4.28: Multiple variant is permitted if the supervariants have a common ancestor.

It is not possible to combine variants that have different corresponding primary types since this would mean that the subvariant was a variant of two different primary types and if this is the case then its type should be a subtype of both of those primary types. If it is appropriate then the corresponding subtyping relationship must be introduced at the primary type level (the primary types must be in a specialisation relationship, or must have a common supertype via specialisation), if there is no specialisation relationship between the primary types corresponding to two variants then they cannot be common supervariants of the same subvariant. In the latter case, construction must be used, construction is less powerful but more flexible, the weaker coupling associated with construction is appropriate to a weaker conceptual relationship (similarity rather than a shared supertype).

The SIR approach ensures that the strong coupling introduced by the variant relationship is only introduced when the corresponding concepts are strongly coupled. This should reduce the fragile base class problem in which changes to a superclass, for the benefits of its clients, have a negative effect on the subclass. Since the supervariant is a generalisation of the subvariant, any implementation associated with the supervariant should, in general, also be applicable to the supervariant.

Multiple Construction Multiple construction can be used to combine existing implementation components. A subclass in a construction relationship may be constructed from multiple instances of different abstractions (or from multiple instances of *the same abstraction*). For example, construction may be used to add the behaviour of scrollbar and title bar abstractions to a window abstraction.

Where multiple construction units provide the same behaviour, *all* appropriate methods will be invoked (with return arguments, if they exist, being thrown away) unless a corresponding method exists on the subclass in which case that method alone will be invoked. This technique can be used to combine multiple methods in a more complex manner. For example, we might have an Meeting abstraction in a calendar application which is constructed from a Notifier abstraction which can notify an interested party when a meeting is modified. There may be multiple notifier objects contributing to the behaviour of the event so that when it is modified multiple parties are notified in the ways in which they have selected (a pop–up alert box, an email, or a message on their telephone answering service). See Figure 4.29.

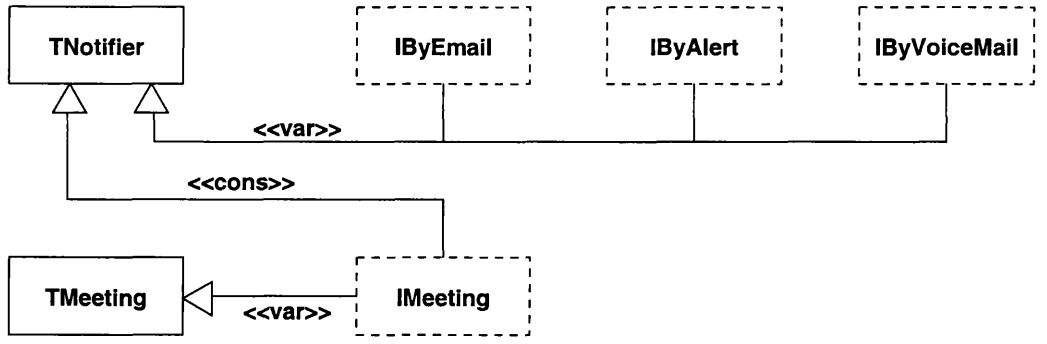

Figure 4.29: Multiple construction is possible from a single superclass.

Multiple View Multiple view would mean that a view instance was a view of multiple objects. For example, a 2-D point being a view of each of its coordinates. This would mean either giving up the concept that a view object inherits the identity of its target (it now has multiple targets) or developing a more complex notion of identity. The SIR model currently only supports single inheritance of views. The utility of multiple views requires further investigation.

Multiple Evolution Multiple evolution requires some interpretation. It could mean that the same adaptation is applied to multiple superclasses, or it could mean that the superclasses are evolved into a single abstraction.

It is useful to be able to apply an adaptation to multiple abstractions — for example, adding an adaptation that reports each method invocation to a debugging version of an application — but it is not really multiple inheritance in the usual sense. It makes sense to define an implementation class with the required behaviour and create an adaptation that is a variant of it in order to apply that behaviour to an existing class. This allows the same adaptation to be applied to multiple superclasses as shown in Figure 4.30.

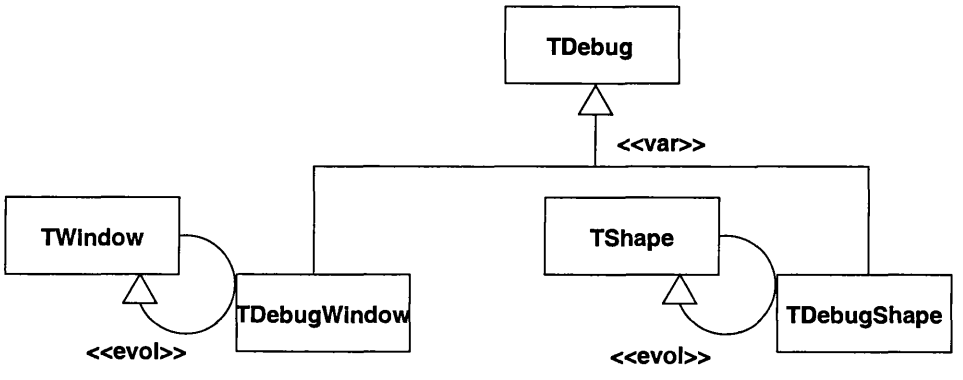

Figure 4.30: Reusing an adaptation.

Evolving two abstractions into a single abstraction is the correct conceptual meaning of an evolution relationship with multiple superclasses. Multiple evolution of this kind is not supported with the SIR model but can be modelled by replacing one abstraction with a view of the other, or by evolving the clients of one abstraction to use the other.

4.9.2 Heterogeneous Multiple Inheritance

As well as homogeneous combinations of inheritance, heterogeneous forms of multiple inheritance — those that involve more than one SIR — must also be considered.

Some combinations of SIRs cannot occur: specialisation, which requires that the subclass is a primary type, cannot be combined with view, variant or construction, in which the subclass is always an implementation class.

The following sections describe the valid forms of heterogeneous multiple inheritance within the SIR model.

Variant and View View is used in combination with variant to support the creation of a variant implementation of one type which is also a view of another type. For example, we can have a variant of a Pair abstraction which is also a view of Point abstraction.

Variant and Construction Variant can be used to implement a type using an existing implementation in the form of a construction unit. The result is a variant that realizes (or partially realizes) its primary type through a construction relationship with an existing implementation. A single variant relationship may be combined with multiple construction relationships in this way. For example, a variant of a meeting abstraction in a calendar application may implement its change–notification behaviour via a construction relationship with a notification abstraction, as discussed earlier.

View and Construction In some cases a view will want to offer some or all of the behaviour of its target to its own clients. This behaviour is achieved by introducing a construction relationship between the view and target. For example, we might have a Person being viewed as a Student where the behaviour of a Person is of direct use to clients of the Student abstraction. The required combination of view, variant and construction is shown in Figure 4.31. View and construction both lead to object–level relationships. In order to achieve the correct behaviour in this situation, the runtime instantiations of the relationships must both have the same superclass instance.

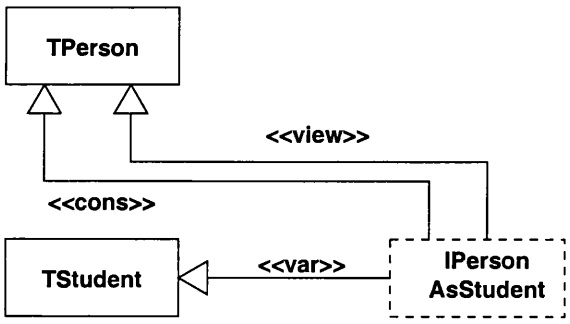

Figure 4.31: Combining view and construction.

View and Specialisation A specialisation relationship must be introduced between a view type and its target type if view instances are substitutable for target instances. For example, if we want to

be able to substitute students for persons then we must introduce the relationships shown in Figure 4.32. This is not multiple inheritance directly, but a combination of two SIRs.

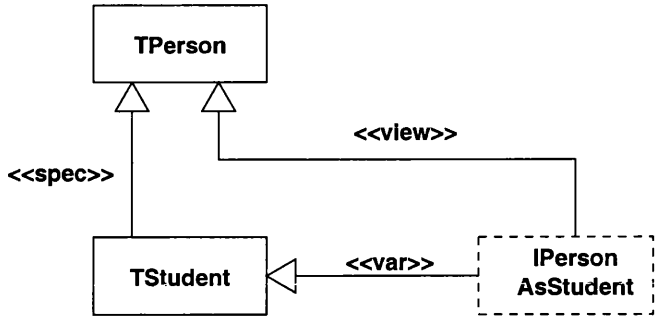

Figure 4.32: Combining view and specialisation.

Evolution and Specialisation Evolution may be used to add a new supertype to a type by combining specialisation with evolution. Multiple specialisation relationships may be combined with a single evolution relationship in this manner. For example, we might wish to evolve a Car abstraction so that it is a subtype of a Vehicle abstraction when we introduce a new Van abstraction. This is shown in Figure 4.33.

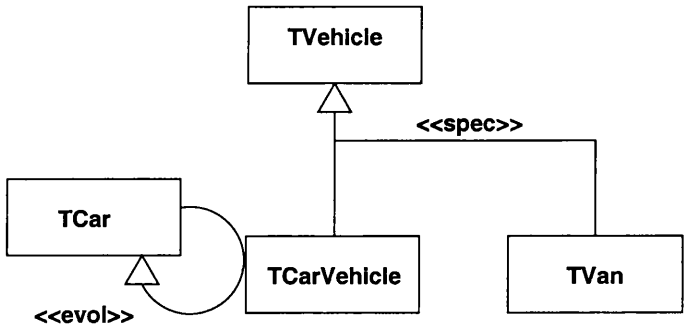

Figure 4.33: Combining evolution and specialisation.

Evolution and Construction/Variant Similarly, evolution can be used to introduce a construction relationship or a variant relationship to an existing implementation class. For example, we might evolve a variant implementation of a string class so that it makes use of an array construction unit. Multiple construction and/or variant relationships may be combined with a single evolution relationship in this manner.

4.10 Relationship with Other Work

The inheritance relationship has provided the impetus for much recent research. In this section we compare this SIR model of inheritance with related work. In addition to comparing the SIR model with other classifications of inheritance, it is also useful to consider the relationship with design

patterns and proposed inheritance–like mechanisms. If the SIR model can explain the semantics of inheritance–like constructs then this provides evidence that the SIR model is more powerful than existing inheritance mechanisms. In addition, the SIR model provides a basis for understanding suggested inheritance–like constructs for programming languages, and if the SIRs were directly supported in programming languages then we would also have an implementation mechanism on which to base higher–level inheritance constructs.

Relationship with Design Patterns The description of five relationships that represent frequently observed idioms in software development has much in common with the notion of design patterns [Gamma et al., 1993]. Like design patterns, the five SIRs document 'best practice' and add to the high–level vocabulary of software design. This is in itself a valuable outcome of the classification of inheritance presented here. However, we go a step further and say that these relationships don't just promote *best* practice in the use of inheritance but that they constitute the *only* way of doing things. That is, all applications of inheritance must fall into one of the five categories or must be achievable using combinations of them.

Meyer's Taxonomy of Taxonomy Meyer describes ten kinds of inheritance [Meyer, 1996] that are appropriate to the Eiffel language. Most are generally applicable to languages supporting inheritance. This taxonomy describes valid and useful forms of inheritance but does not attempt to be exhaustive. Unlike the SIR model, Meyer describes the relationships at a low level, hence the comparatively large set of relationships.

Three high–level classifications of the relationships are described:

1. Model inheritance, reflecting "is–a" relations between abstractions in the model.
2. Software inheritance, expressing relations within the software, with no obvious counterpart in the model.
3. Variation inheritance — a special case that may pertain either to the software or to the model — serving to describe a class through its differences with another class.

Rough correspondences with the SIR model would be that model inheritance corresponds to specialisation and view; software inheritance corresponds to construction; and variation inheritance corresponds to variant. Evolution is not directly supported. Meyer's taxonomy of three high–level relationships is incomplete in comparison with the SIR model since there is no support for evolution. Additionally, the lack of separation of the two very different specialisation and view relationships means that Meyer's high–level taxonomy does not describe fundamental forms of inheritance, as does the SIR model.

Budd/Halbert Classification Budd describes eight general categories of inheritance [Budd, 1997] which are based on Halbert's classification of inheritance [Halbert and O'Brien, 1987]. As with Meyer's taxonomy of inheritance, Budd's forms of inheritance are not intended to be exhaustive and

are based on what is achievable in object–oriented languages. This leads to the following observation: 'it sometime [sic] happens that two or more descriptions are applicable to a single situation, because some methods in a single class use inheritance in one way while others use it in another.'.

This is not the case with the SIRs: each relationship corresponds to exactly one conceptual relationship. Multiple SIRs may be used in combination but they are not merged into a single inheritance relationship. This distinction between the SIR model and Budd's classification is due to the different levels of classification and the different intentions of the classifications: Budd is observing uses of inheritance in practice whereas the SIR model aims to distinguish between *fundamental* forms of inheritance. The SIR model provides a basis for good practice whereas Budd's classification simply observes existing practice.

Correspondence between Budd's classification and the SIR model is as follows: Subclassing for Specification is a form of specialisation; Subclassing for Specialisation straddles the categories of specialisation and variant; Subclassing for Extension and Subclassing for Combination straddle the categories of specialisation and construction depending on whether type or implementation is being inherited; Subclassing for Limitation, Subclassing for Construction and Subclassing for Variance are class–based versions of construction; and Subclassing for Generalization is a form of evolution. In comparison, the SIR model of inheritance is much clearer than the muddled approach that results from observing current practice.

Singh — Important Uses of Multiple Inheritance It is useful to contrast the uses of multiple inheritance described here with the 'four important uses of multiple inheritance' proposed in [Singh, 1994]:

- Multiple Independent Protocols: This covers situations where a class is created by combining completely independent superclasses. Multiple independent protocols can be supported creating a number of views in the SIR model.
- Mix–and–match: Here several classes are specifically created for subsequent combination. These special classes are also known as mixins. Mix–and–match is supported by multiple construction (with specialised variants of construction units being created as appropriate).
- Submodularity: This covers situations where while creating a system, modularity of subparts is noticed and factored out for good system design. Submodularity is supported by the variant relationship.
- Separation of Interface and Implementation (also known as the Marriage of convenience). Separation of interface and implementation is supported by combining the variant relationship with construction.

The SIR model of multiple inheritance therefore subsumes the four important uses of multiple inheritance suggested by Singh. Due to the additional modelling power of the SIR model it has also been possible to identify other valid uses of multiple inheritance as described in this chapter. The SIR model therefore provides a more powerful model of multiple inheritance.

Mezini's Meta–Combiners Mezini [Mezini, 1997] develops a three–layer model of inheritance: behaviour definition, behaviour provision and behaviour combination.

> The behavior definition layer is responsible for the specification of behavior as a set of independent software modules, and the provision layer is responsible to provide clients with functionality. ... The behaviour combination layer represents the additional abstraction between an object and its behavior definition. Its elements, called *metaCombiners*, are responsible for the structural aspects of the behavior of the underlying object and its evolution.

The flexibility provided by this model is also provided by the SIR model. MetaCombiners are similar to variants with construction and evolution being used to control the behaviour of a particular object at a particular time. Just as with evolution, metaCombiners allow 'adjustments' to replace or extend the behaviour of an object when it is in a particular state or when a particular condition holds.

Although powerful, the metaCombiner concept does not provide design–level support for modelling with high–level conceptual relationships, as the SIR model does. The metaCombiner approach can be seen as a possible implementation of much of the SIR model of inheritance since it operates at a lower–level.

Context–Relations Context–relations [Seiter et al., 1998] as found in the adaptive programming paradigm [Lieberherr, 1996] propose a new inheritance–like relationship that allows the behaviour of an object to be altered by its current 'context'. A context object can be attached to an object for the duration of a method invocation, altering the behaviour of that object.

Within the SIR model, this relationship can be modelled using view. If the altered behaviour is required for just a single method invocation then there is no need to retain a reference to the view; the view can be created and the appropriate method called in a single statement.

Context objects are objects in their own right which can be instantiated and applied to multiple objects. Since views inherit the identity of their targets the same view cannot be reused in this way. However, the configuration aspect of the view can be captured in a separate abstraction and a number of views can be constructed from the same configuration instance.

A further application of context objects is to apply them to an object so that all invocations on that object will have the contextualised behaviour. This can be achieved by implementing the behaviour of a variant using construction (evolution can be used if this usage was not foreseen); the object to which all implementation is delegated can then be replaced with the appropriate view of that object. Removing the context involves replacing the view object with the original object.

Although context–relations are a useful technique, it is preferable to model their semantics through the fundamental inheritance relationships provided in the SIR model. Context–relations implement a specific technique whereas the SIRs are based on underlying conceptual relationships and can be combined in various ways to implement many techniques.

Environmental Acquisition The environmental acquisition mechanism [Gil and Lorenz, 1996] allows objects to inherit particular features from their current context. An example taken from [Gil and Lorenz, 1996] is that car door might inherit its colour from the car to which it belongs.

This mechanism can be explained within the SIR model using construction and view relationships. The car door takes some of its characteristics from the car. We can construct the car door abstraction from a restricted view of the car (rather than from the complete car abstraction since the full set of car operations are not appropriate to the car door). In this way the car door can take its characteristics from the car it is currently part of.

As with context-relations, it is preferable to model environmental acquisition in terms of fundamental inheritance relationships than to introduce a new construct with its own semantics.

4.11 Conclusions

In this chapter a new model of inheritance was developed based on five specialised conceptual relationships (the SIRs which were introduced in Chapter 3). The semantics of each of the five SIRs — specialisation, variant, view, construction and evolution— was presented together with a discussion explaining key decision points that emerged in developing the model.

Each relationship was discussed with respect to other relationships that might be considered to achieve a similar result in certain circumstances. In each case, the applicable SIR was highlighted with an explanation of why the other relationship(s) were not appropriate, and in this way, the SIR relationships were shown to be conceptually orthogonal to each other. Each relationship has a rôle to play in building structured systems. The five relationships together provide a foundation for inheritance. In addition, analyses were provided of circumstances where a particular behaviour might be expected to belong to a certain SIR but could in actual fact be achieved used another SIR, or a combination of two or more SIRs. The introduction of the five SIRs also provided the basis for a detailed analysis of multiple inheritance and the valid forms of multiple inheritance were also presented as part of the model.

The model of inheritance presented here makes it much simpler to make good use of inheritance. Much of the wisdom associated with inheritance guidelines and heuristics is embedded in the semantics of the five structured inheritance relationships. Rather than understanding a large number of, often conflicting, guidelines developers need only understand five conceptually simple relationships.

Key results produced during the development of this model include:

- A simplified notion of access in which public, private and protected access modifiers are not required.
- An examination of the Abstract Superclass Rule in the context of the SIR model.
- An analysis of support for delegation in a type-safe environment.

- A detailed analysis of multiple inheritance made possible by the distinction between the five forms of inheritance in the SIR model.

In the next chapter we put the model to the test by showing how it can be used in practice. A number of techniques for using the SIRs in designing structured systems are developed.

Chapter 5

Techniques for Structured Use of Inheritance

This chapter is concerned with providing systematic techniques for using the SIR model of inheritance, as described in Chapter 4, as a basis for the disciplined construction of software systems. A set of inheritance–specific techniques is provided which is intended to augment existing object–oriented (OO) methods.

The SIR model of inheritance provides a more advanced starting point for the description of inheritance–related techniques. The five specialised inheritance relationships provide a higher–level design vocabulary that the usual general notion of inheritance. Additionally, due to the expressive power of the SIR model we can also provide techniques for scenarios that do not have elegant solutions in conventional object–oriented methods.

5.1 Required Architectural Qualities

The purpose of the set of techniques provided here is to provide a structured framework for software design that leads to effective architectures. More specifically, the techniques should facilitate the design of maintainable large–scale software systems which evolve over time and exhibit high levels of reuse. The architectures resulting from software design within the SIR framework are therefore required to exhibit the following characteristics:

1. Systems are readily understandable due to the effective management of complexity.
2. Systems are easily modifiable in the face of changing and additional requirements.
3. Systems generate elements that are reusable, both within the current system and by other systems in the same domain.

4. Systems reuse existing elements that have resulted from domain analysis or from other systems.

A number of qualities contribute to the degree to which a software architecture may exhibit these characteristics:

Variability Variability refers to the ability of an abstraction to be put to multiple uses. It is much harder to reuse a component which is fixed than one that is configurable. For example, a Mouse abstraction in a graphics application exhibits a higher degree of variability if it can be used to represent mice with different numbers of buttons (1–button Macintosh mice, 2–button legacy PC mice, 3–button Unix mice and n–button modern mice), than if it makes the assumption that all mice have two buttons.

Variant and evolution are powerful tools for introducing variability into a system. They also require considerably less forward planning than traditional approaches. This is not to suggest that forward planning is a bad thing but it reflects the fact that there will always be circumstances that were not predicted.

Modifiability Modifiability refers to the ability to change an existing system, not to select between options as in variability, but to actually replace existing functionality with new functionality. For example, changing the default currency used throughout an application from Sterling to the Euro. Modifiability is important when you wish to ensure that the old functionality is no longer in the system.

Note that modifiability can be implemented as variability, by introducing the modification as an option to current behaviour, but this is not safe. If a particular choice is made you cannot guarantee that it will apply throughout the system, other developers are free to (erroneously) choose the alternative in the same system, and it is easy to miss some existing clients when updating them to use the new version. If an element within a system is modified this is not the case. In addition, if something in a system is modified (rather than the new behaviour simply being an alternative to the old behaviour) then it is simpler to reason about the new system.

Extensibility Extensibility is the architectural quality that allows new functionality to be introduced into a system *without modification to existing parts of the system.* For example, a video store application that currently only deals with private customers may need to be updated to handle corporate customers, extensibility is the quality that allows this to be done with the least change to the existing system.

Reusability Reusability refers to the applicability of a software element in a number of different situations. A reusable element should not make unnecessary assumptions, it should allow for variability. For example, by supporting mice with variable numbers of buttons, a Mouse abstraction becomes more reusable. On the other hand, an abstraction that is overly complex will also prohibit reuse.

Reuse Reuse is concerned with actually reusing existing software elements (rather than making them reusable). For example, reusing a Matrix abstraction in the implementation of a Chess Board abstraction. Systems should exhibit high levels of reuse for a number of reasons:

1. Development costs are reduced since there is less need to write new code.
2. Consistency is improved for both developers and users of the system.
3. Improvements to the reused element benefit systems that have reused it as well as the system for which it was originally developed.

Inheritance is supposed to make it easier to build systems with the above qualities, but success has been limited in practice, due largely to the variety of ways in which inheritance mechanisms can be used. The amount of variability, modifiability, extensibility, reusability and reuse within architectures can all be increased by using the techniques that contribute to the SIR framework.

The techniques presented in this chapter are divided into four categories: Planning — techniques that promote future reuse; Variability — techniques that provide alternatives at a particular point in a system; Adaptation — techniques that allow an existing abstraction to be tailored for use in a new situation; and Modification — techniques that allow an existing system to be modified in a consistent manner. This classification is not strict, some techniques could be considered to belong to multiple categories. The range of categories illustrates the utility of the SIR model across the modern software development process.

5.2 Planning Techniques

Planning techniques capture aspects of design that are not required to meet current system requirements but that are valuable outside that context: for reusability outside the current system and in future versions of the current system.

5.2.1 Selecting Features for a New Abstraction

Choosing an appropriate interface for a new type is not simply a matter of including all currently identified operations. It is important to include the right operations when designing a new type. Too few operations and the type will not offer sufficient services to its clients, too many operations and the type will appear too complex and cumbersome to use and, more importantly, because it is over–specified, its chances of being reused (or inherited from) are small. Under a traditional approach, all behaviours that can apply to an instance of an abstraction throughout its lifetime must be included in the abstraction.

The SIR approach offers a more structured way of selecting features for an abstraction. When developing an abstraction that should be reusable it is important to offer the right 'default view'. Offering a large amount of functionality that is not required by a client will make it more difficult for them to see the underlying functionality which they may wish to use. Additional functionality

also means additional overhead which may not be acceptable. On the other hand, if a client does want the extra functionality then they too will be able to use the view of the abstraction developed for this system.

The default view should contain all of the methods that are intrinsic to the abstraction being described. The view of an abstraction required in the current context may not be the default view of the abstraction in the system, the default view should not contain methods that are applicable to a particular client under consideration. The two notions must be clearly separated. This approach avoids class interfaces growing ever more complicated as they are evolved over the lifetimes of several systems but it still provides flexible access to the abstraction.

The complexity exists at the domain level where a large number of interfaces to an abstraction may exist in a reuse repository. Once the required views of an abstraction for a particular system have been selected or developed we have a simple, structured way of using the abstraction at the system level. This is appropriate since views need to be selected only once (or possibly more if a mistake is made or new information becomes available) whereas they are used many times in a system.

Example Consider a video shop system with a Video abstraction, analysis has shown that the video abstraction should have a title, director, producer, list of main actors and a distributor, and that it must be possible to determine whether the video is currently on loan, and if so which customer has the video. It must also be possible to determine how many times the video has been loaned out so that it can be withdrawn after a certain number of uses.

It is important to separate out the intrinsic properties of videos in the video store domain from the properties required when the video is playing a certain rôle. The properties concerned with renting should be separated from the video itself. This is made clear by considering what happens when we realise that the video store must also have videos that can be bought which should not have properties related to renting, and that once a rentable video has been rented a certain number of times it may be sold off at a discounted price.

It is clearly preferable to move the properties concerned with renting to a Rentable abstraction and have videos viewable as Rentables. A Saleable abstraction can then be introduced to handle Saleable videos. Additionally, a video that was once in a Rentable rôle may later play a Saleable rôle. The structure required for this behaviour is illustrated in Figure 5.1.

We get three types — TVideo, TRentable and TSaleable — each of which can be used at appropriate points in the system. We can create video objects with the TVideo type as instances of IVideo. Such instances do not exhibit behaviour concerned with renting and loaning videos, when such behaviour is required a view instance of IRentable or ISaleable is created, as appropriate. Such a view instance allows a video object to exhibit the behaviour required by TRentable or TSaleable, while not polluting the TVideo type with operations that are not appropriate to all videos.

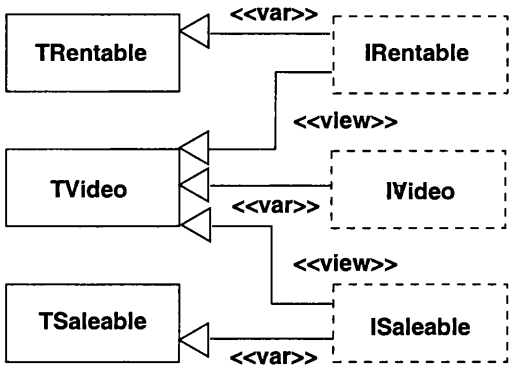

Figure 5.1: A video abstraction with rôle-specific behaviour in separate views.

5.2.2 Reusability of Clients

It is necessary to consider the reusability of clients as well as servers. Often we may wish to reuse an existing client with a new server. For example, a spell–checker defined to operate on text documents could not be reused in the context of spreadsheet or a diagram, even though these abstractions also contain text. A client should only request appropriate behaviour from its servers — in this case that a stream of text can be provided. A server should not be rejected on the basis of not providing non–relevant behaviour simply because it was provided by the original server. The spell–checker is not concerned with the pagination features of the word–processor so it should not be dependent on them.

In the traditional approach no action would be taken to ensure the reusability of servers. A client would directly use any abstraction that provides the required behaviour regardless of what other operations it offers.

The SIR approach is that, wherever a server offers behaviour unrelated to that required by the client, the server type should not be used as the basis for a client–server relationship. Instead a new type must be created to describe the required behaviour, the server is then accessed via this interface. Note that any introduced type must form a valid abstraction. It is not usually appropriate to create an interface for a single method, a set of related methods should be selected. This approach ensures that only useful abstractions are introduced into the system and the proliferation of small, utility types is avoided.

If the new type is a generally useful supertype of the server, which currently does not exist, the supertype should be created and inserted at the appropriate point in the hierarchy (using evolution) to give the relationship shown in Figure 5.2.

If the new type is context–specific, or if it is best specified using an interface that is not compatible with the actual server interface then a view should be used (see Figure 5.3).

Either approach will enable the reuse of the client with any server that can be viewed using the appropriate interface. That is, any instance that can be viewed as a subtype of the interface used by the client. The client may be a client by association or it may be in a construction or other SIR relationship with the abstraction. The same technique is applicable in each case.

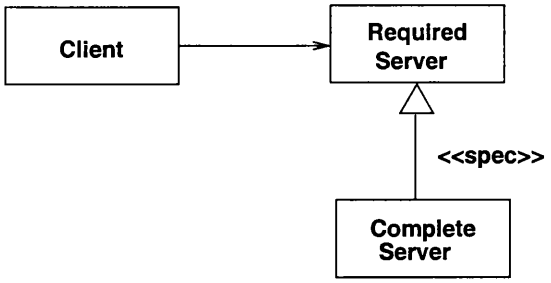

Figure 5.2: Client reusability through abstraction.

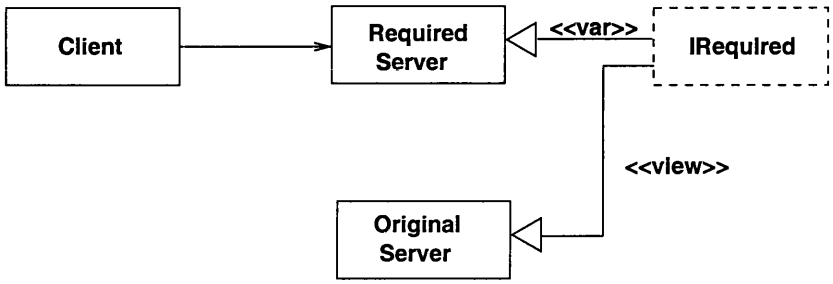

Figure 5.3: Client reusability through a view.

Example Consider a student records system with a Student abstraction where each student has a name, contact details, an identifier, and an associated course. When writing a letter to a student it is necessary to use their name and contact details in order to address the letter. The system therefore contains a Postal Address abstraction which is associated with a particular Student. In fact, the Postal Address abstraction only requires the name and address of the student, it is not specific to the concept of students at all. In order to be reusable the Postal Address abstraction should be implemented in terms of a Contactable (or similar) abstraction through which the Student abstraction can be viewed (Figure 5.4). Construction can now be used since the properties of the Contactable abstraction are also properties of the Postal Address abstraction (in other cases a simple clientship relationship would be appropriate).

Using this approach the Postal Address abstraction can be reused in scenarios where other Persons or Organisations must be contacted by letter.

5.2.3 Separation of Reusable Implementation from Variants

If implementation is placed in a variant then it is tightly coupled to the type it implements, if the requirements of clients of the type change then the implementation in a variant will also need to change. If the variant is used as a superclass in a construction relationship then the subclass will be affected by any changes that are made to that variant. Since the subclass has different clients to the superclass, the changes may have a negative affect on the subclass.

Traditionally, implementation would be reused via subclassing. This can lead to the fragile base class problem (discussed in Chapter 2) if the superclass is modified to suit its direct clients.

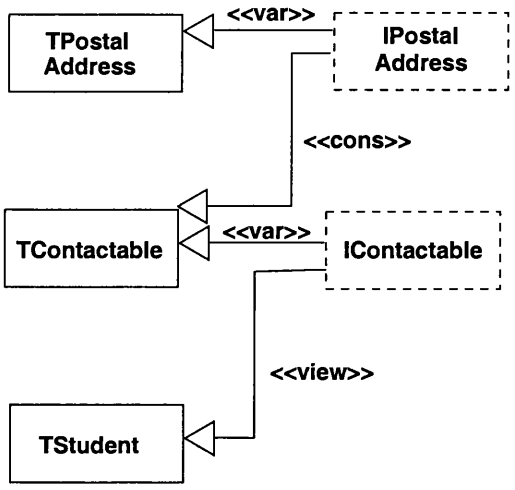

Figure 5.4: Constructing a postal address from the contactable aspect of a student abstraction.

The SIR approach to avoiding this aspect of the fragile base class problem is to distinguish between reusable and variant–specific implementation. When implementing a variant the developer should consider how much of the implementation is specific to the variant being implemented (and any subvariants of it) and how much is more generally applicable. Generally applicable implementation should be placed in a construction unit and the variant should be constructed from it. In some cases a variant may be completely implemented in this way. Any variant–specific implementation that is not intended to be made available for reuse can be placed inside the variant itself.

The variant that requires the reusable implementation, and any other classes that reuse that implementation, now have the same relationship with the implementation. This situation is greatly preferable to inheriting from a superclass that models a particular abstraction just because it *currently* offers a useful set of behaviours. Note that this technique is an extension of the concept of separating implementation from interface which is well understood. We go a step further and separate *both* interface and implementation from the description of a set of instances: the variant.

Example Consider a Person abstraction, one of the aspects of which is a date of birth. Rather than implementing the date of birth and related operations as part of the Person class we should realise that the date of birth is simply a Date; it is the association with a person that makes it a date of birth. The implementation of the Date should be factored out into a separate abstraction so that it can be reused in other implementations, or directly used as a Date abstraction as shown in Figure 5.5.

Operations such as age and birthday which may be implemented in terms of the date of birth properly belong to the Person abstraction. A change in say, the way that birthdays are formatted for printing, will not affect other clients of the date abstraction or other abstractions that are constructed from the date abstraction.

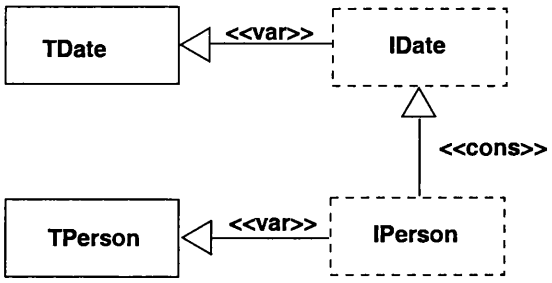

Figure 5.5: Date functionality is factored out into a separate abstraction.

5.3 Variability Techniques

Variability techniques support the introduction of alternate functionality at particular points in a system. Variability enables systems to support alternate functionality without introducing new abstractions, existing abstractions can be used in different ways.

5.3.1 Construction–Time Variant Selection

The client does not always have sufficient information to determine which variant should be used to instantiate a new object. For example, the appropriate variant of a printing abstraction may depend on the characteristics of the physical printer that is to be used for printing. A client creating a printing instance to handle the printing of a document does not wish to know which variant is appropriate for a particular printer.

In the traditional approach, the Factory Method pattern [Gamma et al., 1995] can be used to create an instance of a class or one of its subclasses. The client invokes the factory method rather than directly calling a specific class constructor. A factory method could be used to instantiate an appropriate printing object in the above example.

The use of a factory method also applies within the SIR model. All that a client knows within the SIR model when it calls a constructor is that it will get back an object meeting the specification guaranteed by that constructor. There is no obligation for that object to be of a particular class. This allows the same constructor on a super–variant to return instances of different sub–variants depending on arguments or on the current context (see the example below). It is possible for any constructor to be evolved so that it returns instances of one of its subclasses, rather than an instance of itself. The client only has access to a list of constructors; information on how those constructors map onto implementation classes is hidden. (In a language supporting the SIR approach it should not be possible to distinguish between a factory method and other constructors.)

Example Consider a rôle–playing game system in which there are various kinds of monster, represented by a Monster abstraction. At various points in the game the system will create a new Monster. There are different kinds of monster with different characteristics (strength, special actions, etc). The kind of monster to be created at any particular point in the game is not predefined

and depends on a number of factors including the current location, the level of difficulty, the kinds of monster recently seen and a random factor.

This situation can be modelled by having a variant of Monster with a constructor which selects between subvariants corresponding to the different kinds of monster. The constructor would be placed on the IMonster variant in Figure 5.6. Conceptually, the action of creating a new Monster in the system is dependent on a number of factors. Some of these factors may be controlled via constructor arguments and others may be obtained by the constructor from its environment.

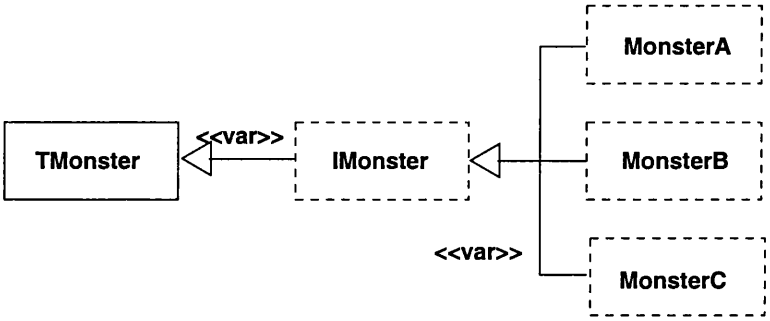

Figure 5.6: A constructor on a supervariant may return a subvariant instance.

5.3.2 State–Based Abstractions

Some abstractions have distinct behaviours in different states. It is often natural to separate out state–specific behaviours into a distinct abstraction. Such abstractions cannot be subclasses of a general class because instances need to move between the classes dynamically. For example, we might have a Bank Account abstraction that behaves differently when it is in credit from when it is overdrawn.

Traditionally, the different states would be handled by conditional statements within each method. Under this approach it is difficult to get an understanding of the complete behaviour of an object when it is in a particular state. An improvement on this situation is the state pattern which implements state-specific behaviour in separate classes which are subclasses of a state class. A wrapper class forwards method invocations to a state-object appropriate to the current state, and access to shared methods (and therefore data members) is achieved by state instances referring to the wrapper instance that makes use of them.

Using the SIR model, evolution can be used to build up the implementation of an abstraction in stages. These stages can correspond to states. The use of evolution may seem strange at first. In fact it is the natural relationship to model state based behaviour because you are adding information and evolving an existing concept to take that information into account. The evolution relationship is concerned with extending the description of an existing concept. The variant relationship, in contrast, is concerned with introducing points at which realizations of the same type can differ.

Using the evolution relationship to model state–based behaviour provides a powerful mechanism. It is possible to describe the core (non-state specific) behaviour in a superclass and then describe

state-based behaviours in a set of subclasses, each of which depends only on the superclass. There is no need to define a class which combines all of the behaviours, they are each added to the superclass using evolution. Selection of the appropriate method for a particular invocation may be based on a state than is derived from the current value of an object or, on a state that is explicitly recorded. Language support could be used to provide automated selection on either basis.

Example Consider a vending machine system such that when a customer presses a button an instance of a Selection abstraction handles the customer's request. The behaviour of the selection will depend on whether the requested item is available or not and also whether or not the current item is the last item (when it is necessary to light the sold-out indicator after the item has been dispensed). The behaviour when the selection is available and not the last item can be seen as the default behaviour. This behaviour can be extended using evolution to get the behaviour for the last item and overridden to get the behaviour when the item is sold out. This is shown is Figure 5.7. The evolution relationship allows the realization within ILastItem and IEmpty to extend the ISelection variant without modifying it directly. This structure allows the implementation of the vending machine to be developed and understood in stages, thereby reducing complexity.

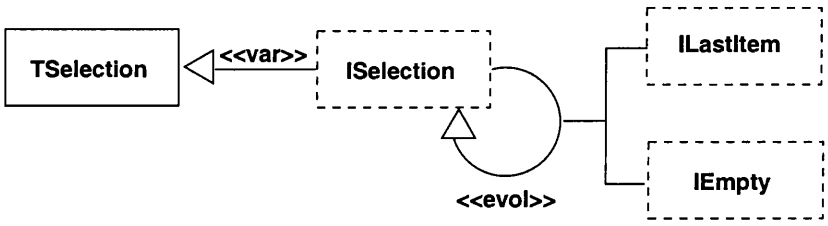

Figure 5.7: Using evolution to build up state-based behaviour.

Note that this approach supports extension when a new state is identified. Suppose the machine is upgraded with the ability to call back to base when a particular item needs restocking. A new state could be added corresponding to a selection that is three–quarters empty. This would extend the default behaviour by sending a message back to base.

5.3.3 Dynamic Realization Variation

The realization of some objects may need to vary at run–time. For example, as the elements of a data–structure vary the algorithm required to sort those elements, in the least time, may change. Traditionally, inheritance offers no support for dynamic implementation variation.

Within the SIR model variant supports creation–time selection between implementations and should be used when it applies. If the dynamic change in behaviour is due to the internal state of an abstraction then the State–Based Abstractions technique described above should be used. However, change of internal state is not the only reason to require behaviour to vary during the lifecycle of an object. The obligations of an object may change with respect to its interactions with external objects. (Of course, there is a fine line between distinguishing between internal state and relationships with other objects, this is dependent on context and the developer's judgement.)

In such circumstances the construction relationship is valuable. Construction offers a conceptual relationship between abstractions but allows the actual instances in an instantiation of the relationship to vary at run–time. In order to simply switch between implementations a variant is created which is constructed from an implementation type of which the alternate implementations are variants. The actual instance from which the object is constructed can be changed dynamically to move between different implementations, the subclass must provide mapping between the different implementations.

Example Consider a Matrix abstraction in a numerical computation system. It is necessary for clients of the abstraction to be able to switch between dense and sparse representations of the matrix at different points in a calculation.

This can be achieved using construction from an abstract MatrixRep implementation class with Dense and Sparse subvariants. A variant of the Matrix class is then constructed from the MatrixRep class and can dynamically vary its representation at the request of its clients. This is shown in Figure 5.8. Note that it must be possible to convert between the two implementation types, this may be achieved by each implementation variant having a constructor taking an instance of the other implementation variant as an argument.

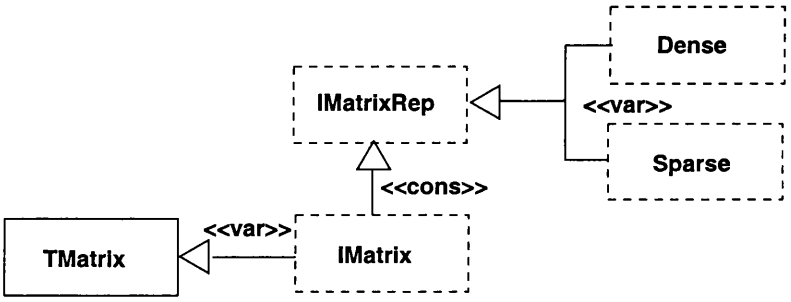

Figure 5.8: Dynamic variation of realization.

5.3.4 View as an Alternative to Multiple Dispatch

In some circumstances, multiple dynamic argument types are required to select the appropriate version of a method for a particular invocation. For example, we might have abstractions representing various chemicals; when combining two chemicals the result is dependent on the types of both of the chemicals. Multimethods may need privileged access to multiple arguments, for example, chemical reactions may depend on details of chemicals that are not made available via their public interface.

As discussed in the previous chapter, the SIR model does support the notion of multiple dispatch. However, many programming languages do not and additionally many problems that appear to require multiple dispatch can actually be modelled using the view relationship. Although the SIR model supports multiple dispatch, appropriate modelling, using the technique described here, removes the need for multiple dispatch in many cases.

Many object-oriented examples of multiple dispatch occur when an argument to an operation may be one of a number of related types, possibly including the receiver type itself. In such situations additional work may be required to get the argument type into a form that is appropriate for performing the required operation on the receiver. This behaviour does not belong on the types of the arguments since it is only applicable in the context of the receiver type.

The SIR approach to handling this form of multiple dispatch is to create views of the argument types that turn them into a form suitable for the receiver. In many cases the receiver itself is one of the argument types and the aim is to convert the arguments to the same type as the receiver.

Example A commonly cited example of multiple dispatch is binary operations across the real and complex numbers. In an object–oriented context this often means being able to add a real number to a complex number and possibly, a complex number to a real number — as well as being able to add a real to a real and a complex to a complex. In each case the result must be of the same type as the receiver since the receiver will be modified by the operation.

Rather than adding an operation to the complex number abstraction which accepts a real number and treats it as though it has a zero–valued imaginary part, the SIR approach is to create a view of real numbers that allows them to be treated as complex numbers. This means that the operation on the complex number abstraction for adding complex numbers can be used. This solution is illustrated in Figure 5.9.

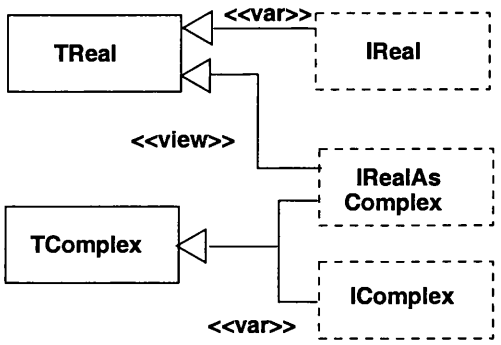

Figure 5.9: Viewing a real number as a complex number.

Similarly a view in the opposite direction can be taken ignoring the imaginary part of a complex number to get a real number that can be added to another real number. This approach makes the relationships between the types very clear, clients must view a complex number as a real number before adding it to a real number.

A similar effect can be achieved by 'casting' between types in C++ and other languages. The SIR approach has the advantage that the type that you are casting from does not need to know anything about the type you are casting to.

5.3.5 Matching

Matching is a mechanism which supports polymorphism for types that are parameterised by type of the receiver. For example, an equality method defined on a point class will accept only coloured points when it is inherited into a coloured point subclass. In order to write code that will operate across points and across coloured points, but not across a mixture of the types, matching is required. Matching is allowed in the SIR model but to support effective modelling we must be able to use matching and subtyping forms of specialisation across related abstractions. For example, there may be some operations, such as location, for which coloured points should be substitutable for points at the instance level (requiring subtyping), and others, such as equality, for which matching is appropriate. Matching is not found in mainstream languages and therefore techniques to handle it during modelling have not been required.

Type systems that allow matching and subtyping polymorphism introduce a separate type to support this behaviour. Matching can be achieved through one type while subtyping polymorphism is achieved through another. In other words, the matching type must be used to achieve method invocations in which the relationship between the dynamic types of the arguments (including the receiver) is significant in selecting the correct method, typically, two or more arguments must have exactly the same type. Conceptually, we would say that matching is used when we wish to rely on the substitutability of a set of typed objects, rather than the more usual instance–level substitutability. It is useful to model the matching type separately from the subtyping (or basic) type, at the design level, to enable more precise modelling.

The behaviour that can be achieved through the basic type (used for subtyping polymorphism) can also be achieved through the matching type (used for matching). Behaviour that requires matching cannot be achieved using the basic type. This means that the two types can be related by specialisation. This is shown in Figure 5.10. Note that a constructor on the matching type must be used to instantiate an object if it is to be used within a matching context.

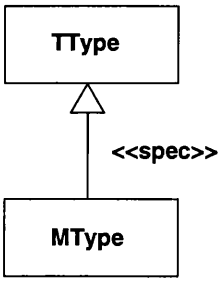

Figure 5.10: The basic and matching types are related by abstraction.

Matching is only important when more than one type shares the operations that require matching. In this case there is a specialisation relationship between the basic types (to provide subtyping polymorphism) and a specialisation relationship between the matching types (to provide matching polymorphism).

Example The relationship between points and coloured points is a classic example of the need for matching. Coloured points should be subtype substitutable for point for some operations (x–coord, y–coord) and matching substitutable for other operations (equality). The required relationships are shown in Figure 5.11. If only traditional subtyping substitutability is required then the TPoint and TCPoint types can be used, operations that require matching, such as equality, will not be available via these types. If matching is required then the MPoint and MCPoint types must be used, these types support matching but introduce restrictions on assignment in order to maintain type safety.

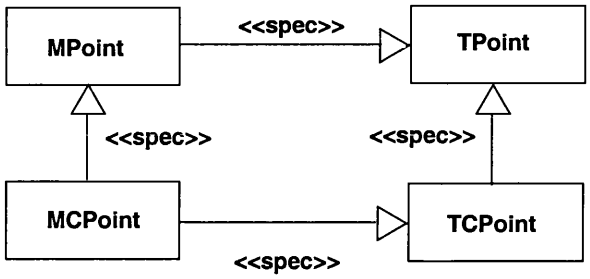

Figure 5.11: Point and coloured point abstractions with subtyping and matching polymorphism.

5.4 Adaptation Techniques

Adaptation techniques allow abstractions to be put to new purposes whilst their contracts with existing clients remain unchanged. Adaptation techniques support reuse within and across systems.

5.4.1 Interface Mismatch

It frequently occurs that an existing component is required to perform a rôle for which it has appropriate behaviour but for which it was not designed and does not offer an appropriate interface. This may occur, for example, when a third–party library is to be reused but does not offer an interface that is consistent with the current application.

The traditional approach to resolving this situation would be to use the Adaptor pattern [Gamma et al., 1995]. In practice this can mean one of two things. Either a new abstraction is created which forwards requests to an instance of the existing abstraction, or a subclass of the original abstraction is created which adds the new interface and implements it in terms of the inherited implementation.

The former approach also works for subclasses of the existing abstraction since it is possible to forward method invocations to a subclass instance as well as a superclass instance. However, identity is not preserved and substitutability does not result. Additionally, this approach does not work if access to internal methods is required to provide the required interface (either at all or in an elegant or efficient manner).

The latter approach preserves identity but does not provide the modified interface to subclasses of the original abstraction since they do not inherit from the new subclass. Additionally, this approach does not prevent the old interface and the new interface from being mixed by clients. Implementing

the new interface in a subclass does allow access to the internal representation of the superclass if it is required to implement the new interface.

Within the SIR model, it may seem that evolution is appropriate here since it would allow us to change all instances of the class. In some cases evolution may be appropriate, but in general it is not the right solution. This is because if we evolve the server, existing clients, such as those in the reused third party library, will not be able to interact with the abstraction.

The problem is that we do not really want to change the existing interface of the component at all. We simply wish to provide it with a further set of operations for use in certain contexts. This can be achieved by providing a *view* of the component to be used by the client. In practice an approach close to this is sometimes used to solve such problems: a separate 'wrapper' object is introduced (as described above). The advantage of describing the relationship as a view is that we now have a conceptual basis for the introduction of the extra entity, rather than a simply syntactical one. For example, CASE tools should be able to show all possible views of a particular abstraction; this is quite different from showing all possible *clients* of an abstraction.

The view approach means that the new interface is only used by clients that need it. Any existing interactions between the component and its servers are on the same terms as before. If the view instances need to be substitutable for target instances then an additional specialisation relationship between the target type and the view type is required to achieve this. This approach clearly illustrates the intention of the software developer, unlike the adaptor pattern with inheritance in which subtyping may simply be a side–effect.

Example Consider matching an existing circle class from a third–party library (ABCCircle) with an existing client requiring circle objects that conform to type TCircle. The problem being that the circle component offers a method for determining the diameter of the circle whereas the client requires a radius method.

We can create a view of the circle class that offers a radius method (and any other methods that are required to conform to TCircle). The view can simply implement the radius method in terms of the existing diameter method. This gives the abstraction structure shown in Figure 5.12.

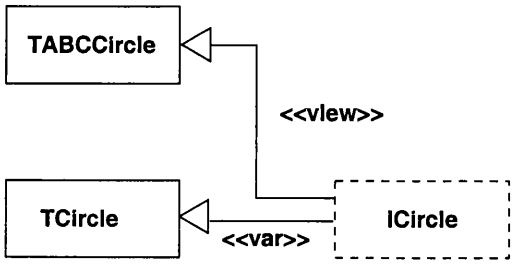

Figure 5.12: Solving an interface mismatch using a view.

The new part of the system must instantiate the appropriate view of each circle as it is passed to the client.

5.4.2 Context–Specific Behaviour

An abstraction may be required to exhibit new behaviour that is only applicable in a particular context. That is, the behaviour is subjective and is a result of the interaction between the basic abstraction and a particular set of clients. For example, some Cars may be used as Taxis, requiring them to have a Taxi ID, and Taxi–related behaviours, such as being permitted to use special Bus–and–Taxi lanes.

The traditional approach would be to use inheritance to create a subclass with the additional behaviours, in this case, a Taxi subclass of Car. The new subclass would need to be used for any objects that may be required to exhibit the subjective behaviour. This solution is inadequate because it requires that all objects that *may* get used in a particular way exhibit behaviour specific to that context.

A further disadvantage of storing subjective state with an object is that there can only be one instance of that state per object; for example we might have an examiner's view of an exam paper in which a grade is assigned, and different examiners may wish to assign different grades to the same exam paper.

The SIR approach uses a view to provide subjective behaviours. Views can be added to existing abstractions without modification to the existing abstraction and without affecting existing clients of that abstraction.

Example Consider a Course Members abstraction which models the list of students on a particular course within a learning management system. Various views of the members of a course are required within the system: the students who are currently failing the course; the students who have not paid their fees; the students in alphabetical order; the students who have not yet submitted an overdue piece of work; and so on. It is likely that each of these views will take the form of an iterator. That is, rather than requiring an actual list of students, clients will require an abstraction that lets them move through the list requesting the next relevant student until they have all been considered.

Offering all of these iterators as part of the Course Members abstraction would clearly complicate it and complicated abstractions are difficult to use and are unlikely to be reused. Additionally, such an approach would not allow the same iteration to be performed multiple overlapping times. For example, one part of the application may be iterating through an alphabetical list assigning exam numbers to students, while the course tutor is trying to assign students to groups for a project. Moving the 'cursor' in one view would also move it in the other, this is clearly not appropriate.

Instead of adding the iterator functionality to the Course Member abstraction we create view abstractions corresponding to the various kinds of iterator (Figure 5.13). A TStudentIterator type is introduced to represent iterators over lists of course members, the different kinds of iterator — Alphabetical, Overdue Work, Failing — are introduced as variants of TStudentIterator since they will all be accessed in the same way. To show that each of the iterator variants are views of TCourseMember, we give them a common supervariant, IStudentIterator which is in a view relationship with TCourseMember.

The view approach allows multiple views to be developed and for multiple instances of those views to be created. We can have multiple instances of an alphabetical view iterator, one being used by a course tutor, and one being used to assign exam number to students. Moving the cursor in one view will not move the cursor in other views.

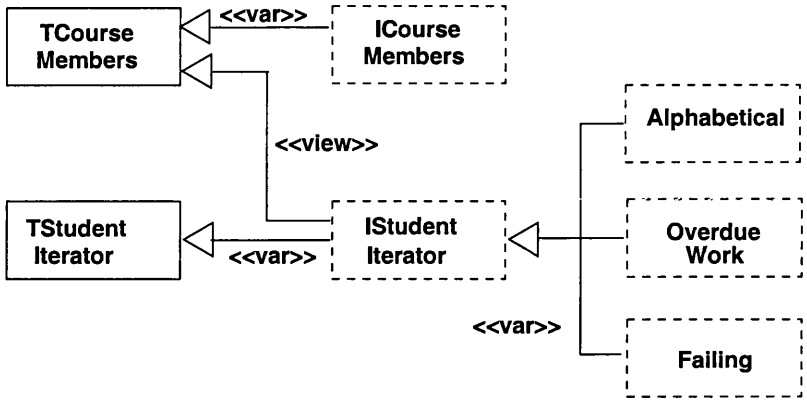

Figure 5.13: Multiple views of a the Course Members abstraction.

Iterators often require privileged access to the abstraction they are iterating over. If this is the case then this can be supported by specializing the view relationship so that the iterator variant views a specific Course Members variant. This should be avoided if possible since it introduces a stronger coupling between the target and view, requiring that if a new Course Members variant is introduced then corresponding iterator views must also be introduced.

5.4.3 Programming–by–Difference

In this scenario, an existing class supports most of the functionality required by a new class which is not behaviourally substitutable for the existing class. The behaviour of the existing class requires specialisation in order to support the behaviour required by the new class. For example, we might have a class which displays error messages, a new class needs this behaviour, but needs error messages to be logged as well as displayed.

Traditionally, this problem would have been approached using private inheritance (also known as implementation inheritance or inheritance for reuse), with superclass methods being overridden as required. This approach has been much criticised since it leads to inheritance hierarchies that are not based on clear conceptual relationships and are therefore difficult to understand and maintain.

The SIR approach does not offer a single inheritance relationship that will achieve the required semantics in one step. Instead the relationship must be broken down into its constituent components: a variant relationship, which achieves the specialised behaviour, and construction which allows the subclass to be constructed from the specialised abstraction.

Note that variant requires that the subclass is fully conformant with the type being realized so only conformant overriding can be achieved in this way. The variant is therefore conformant with its superclass and represents a specialised realization of the same concept, and changes to the superclass

are less likely to cause problems in the subclass because the subclass shares its behavioural semantics rather than just its implementation.

The tailoring of existing methods to suit new uses is clearly separated from the reuse of existing methods. The resulting structure, shown in Figure 5.14, is based on conceptual relationships rather than on the goal of low–level code reuse. If the SIR approach appears complex it is because the modelling problem is complex; it is not surprising that trying to model programming–by–difference with a single inheritance mechanism leads to confusion. The SIR approach clearly separates the distinct inheritance relationships that occur in programming–by–difference.

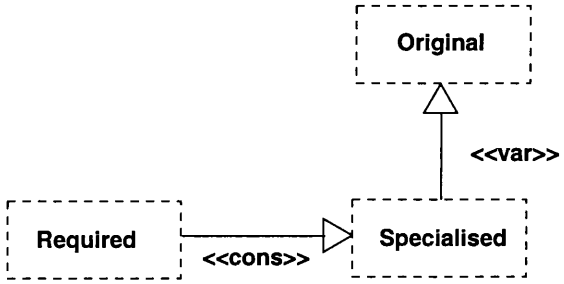

Figure 5.14: SIR version of programming by difference.

Note that by creating a new variant with the modified behaviour we have also made that behaviour available (via construction) to other classes, without them being in any relationship with the subclass. Using the traditional approach reuse of the modified behaviour would have required inheritance of the subclass, or forwarding to one of its instances. If the two classes are not conceptually related this causes further problems.

As an alternative to the variant relationship, it may be appropriate in some cases to use evolution to modify the existing variant. This is appropriate when the required modification is a generalization of the existing behaviour. The default behaviour after evolution should be the original behaviour with an additional mechanism for requesting the new behaviour.

Sometimes it is necessary for the specialisation of the original abstraction to have access to subclass methods. In this case we introduce a construction relationship between the specialised subclass and the class that reuses it. In other words, we have construction relationships in both directions with each abstraction using behaviours of the other in order to implement its own behaviours. This corresponds precisely to the conceptual relationship between the abstractions.

Example Consider an implementation class which performs a notification action when triggered, and also performs related actions such as maintaining a transaction log and resubmitting failed notifications at a specified interval. It is now necessary to implement a web monitoring class which produces reports based on web logs. One of the responsibilities of this class is to email the webmaster if one of a number of error conditions is recognised.

The notification class provides much of the required behaviour, but the web monitoring class must override the notify method to email the webmaster and must invoke the trigger method from methods

that identify error conditions. The new version of the notify method will require access to the web monitoring class for an elegant implementation — for example, the email address of the webmaster and the details to be put into the message will be found in the web monitoring class. This is achieved using construction from a variant of the notification class which, is constructed from the web monitor itself. This example is illustrated in Figure 5.15.

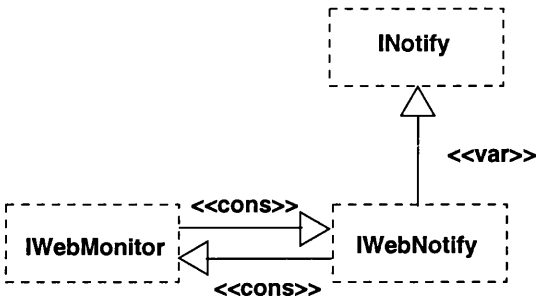

Figure 5.15: Web monitoring class constructed from a specialised notification class.

To reduce the coupling between these applications, the construction relationships could be specified in terms of secondary types describing only the behaviour that is depended upon in each case. The IWebMonitor and IWebNotity variants would then need to be subvariants of the corresponding secondary types.

Note that the relationships between the classes are much clearer than they would have been if a traditional inheritance mechanism had been used to combine the semantics of the separate relationships. Future understanding and modification of the structure is simpler within the SIR model.

5.5 Modification Techniques

Modification techniques allow a particular part of a system to be modified to introduce new behaviour. Modification should be localised and should involve a single abstraction, or a small number of closely related abstractions.

5.5.1 Extending an Existing Type

An abstraction is sometimes found to be missing some key functionality that was not required in another system, or in an earlier version of the current system. For example, a company administration system may have been designed without functionality for retrieving previous job titles of an employee. When it is realised that this functionality is required, for producing employment histories for employees, it must be added.

The traditional solution to this problem — especially when the type to be modified is contained in a third party library — is to use inheritance to create a subclass with the required behaviour. Clients requiring the new behaviour must create instances of the subclass. The problem with this solution is that instances may be created in existing parts of the system, or in reused components, which do

not have the additional behaviour. This is not the required semantics (although it may suffice in some limited situations).

The SIR approach is to use evolution to evolve the original abstraction (and its variant implementations) to support the new behaviour. In systems using the developed adaptation all instances of the abstraction will have the new functionality. The adaptation can be selected on a per–system basis. Additionally, both new clients, and evolved versions of existing clients, can make use of the new behaviour. The SIR approach also reduces the number of primary types in the system.

Example Suppose we have a simple calculator class which allows you to perform operations on the current total (multiply by 2, add 1, reset, etc). Now we would like to reuse that abstraction but we would like to be able to undo operations.

The introduction of an undo operation does not interfere with existing clients and may even be of use to them as their systems evolve. We do not disallow the possibility of systems that prefer to use the simpler functionality (perhaps for efficiency or safety), therefore the undo adaptation may be selected on a per system basis.

We can use evolution to add an undo operation to the calculator type (as in the Command pattern used in Eiffel). This will, of course, initiate a change in any implementations of the calculator which can also be achieved using evolution. We create an adaptation which adds after methods to the methods that execute a command. We must also implement the undo method by executing the more recent command and adjusting the list of commands appropriately. This solution is shown in Figure 5.16.

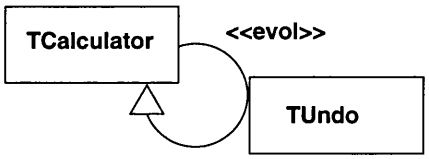

Figure 5.16: Using evolution to extend an abstraction.

Note that this solution is not interchangeable with one that uses a view to add undo functionality. In such a case the undo operation would only apply to operations that are carried out using that view. The introduction of a variant with the behaviour would not work either: the undo behaviour requires the extension of the calculator type as well as its implementation(s).

5.5.2 Generalisation by Evolution

This technique applies when an existing abstraction is too restrictive for use in a new situation. Had the abstraction been designed with the extended functionality then it would have been useful in both the new and old situations. For example, consider a clock abstraction which uses a 12–hour clock with am and pm; the clock cannot easily be reused in a context where switching between a 12–hour and a 24–hour clock is required. If the original abstraction had offered both 12–hour and 24–hour versions then it would have been useful to both sets of clients.

Traditionally, the original abstraction would be subclassed to add the more general behaviour. The original abstraction would still be used by existing systems, whereas new systems would use the new more general abstraction. The weaker functionality offered by the superclass is actually subsumed by the subclass. The superclass is superfluous as a distinct abstraction, existing systems would have used the more general abstraction has it been available.

The SIR solution is to use evolution to extend the abstraction in question so that it can fulfill the requirements of both old and new clients. Using evolution allows the development of a single abstraction rather maintaining a weaker (and superfluous) version and introducing a more powerful version. However, we still get the benefit of describing the more powerful version in terms of its differences from the existing weaker abstraction, leveraging existing development.

With tool support the open/closed principle can be adhered to and clients of the abstraction need never see the less restrictive version. The old version of the abstraction still provides the base for the new abstraction so we make use of existing design (and realization where appropriate) gaining leverage from the existing implementation.

Example Suppose we have a type for formatting error messages, and corresponding variant implementation that operates on the ASCII character set. A new application wishes to make use of this abstraction but it must cope with languages that cannot adequately be represented in ASCII — UNICODE is required.

We can evolve the formatting type and class to ones which do not restrict the character set to ASCII, but to a type containing the operations required on a character set (this may be a new type through which both ASCII and UNICODE can be viewed, or it may be an existing common supertype of the two character sets). The new class contains all implementation that does not require knowledge of the particular character set in use. This may require reimplementing some methods. The ASCII character set is then used by default, to satisfy existing clients and operations are provided to select alternate character sets. This approach is illustrated, for types, in Figure 5.17.

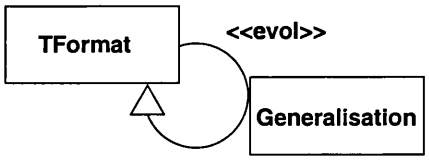

Figure 5.17: Generalising a type by evolution.

5.6 Conclusion

We have provided a number of techniques which illustrate how the SIR model can be used to develop software systems that exhibit desirable architectural qualities such as adaptibility, modifiability, extensibility, reusability and reuse. These techniques supplement and expand upon the specifications

of the SIRs as found in the SIR model (Chapter 4) and represent best practice within the SIR framework.

A number of modelling scenarios were presented along with the traditional approach to solving, or partially solving, the problem and a detailed description of the SIR approach. Simplified but realistic examples illustrate how the techniques can be used in real modelling situations. Improvements gained by using the SIR approach illustrate the controlled power of the SIR model of inheritance.

The variety and complexity of the modelling scenarios considered in this chapter provides justification for the claim that the SIR model can successfully replace — and improve upon — the traditional, overloaded concept of inheritance. The SIR approaches to the scenarios show that the SIR model of inheritance is both powerful and structured: it leads to flexible structures that have a strong conceptual basis.

Chapter 6

Case Studies: Applying the SIR Framework

In the preceding chapters we have built up the SIR framework resulting in the SIR model of inheritance developed in Chapter 4 and the set of techniques described in Chapter 5. In this chapter, the SIR framework is put into practice to demonstrate how the SIR model of inheritance can replace, and improve upon, the existing overloaded and confused notion of inheritance. Through a series of case studies we show that the SIR framework combines structure and modelling power to enable the development of conceptually–meaningful designs which exhibit desirable architectural qualities including reuse, reusability, variability, modifiability, extensibility and adaptability.

Before considering the design of a system in its entirety, we address two well–know modelling problems with the aid of the SIR framework.

6.1 Restricted Subclasses: The Square/Rectangle Problem

The square/rectangle problem was presented in Chapter 2 as an illustration of the inadequacy of an intuitive understanding of inheritance. Square/rectangle is the best known example of a problem that occurs when a subclass is defined as a restriction of its superclass: the subclass may not be fully substitutable for its superclass. We have this situation when a rectangle class, typically combining interface and implementation details, is subclassed to create a square class. Squares are defined by placing a constraint on rectangles: the two dimensions must be equal. This causes problems because operations that apply to rectangles (such as stretch–x and stretch–y) do not result in squares when they are applied to squares.

It is well understood that there is no single solution to this problem. The appropriate way to model the relationship between squares and rectangles depends on the subject domain in which the abstractions occur and on architectural concerns such as trying to increase reuse and reusability.

The SIR model does not provide a universally–applicable solution to this modelling problem (we should not expect to find one since there is no single relationship that can model all situations), but it does help us to distinguish between and identify the possible solutions, so that we can choose the most appropriate one for a particular situation.

The following clear analysis of the problem is made possible by the existence of the SIR framework since the five SIRs provide precise semantics of inheritance.

6.1.1 Time Efficiency Model

Suppose that the goal is to make use of the constraints that apply to squares for reasons of time efficiency. In this case, square objects do not need to be guaranteed to stay square but when an object is square it should make use of that fact for efficiency, that is, square instances must make use of their constraint to provide efficient implementations of operations such as area and perimeter. Rectangle instances in the same system will need to use less efficient methods. In this case, there is no need for clients to be able to distinguish between squares and rectangles, a rectangle operation on a square may result in a rectangle, this means that we have no requirement for a separate square primary type.

Since the same instance may move between states where it is a square and states where it is not, we require a variant of the rectangle that describes instances that will use square methods when they are square and rectangle methods when they are not square. This could be implemented using conditional logic on the current state but it is simpler to understand complex state–based behaviour if each state is described by a separate class.

The state–based abstractions technique of Chapter 5 (see Section 5.3.2) is applicable in this scenario. Operations, including read–only operations, that apply to all rectangle instances can be placed in a rectangle variant and we can then use evolution to add two state–specific adaptations of the rectangle variant: one representing rectangles that have square values and the other representing rectangles that have non–square values. The resulting structure is shown in Figure 6.1.

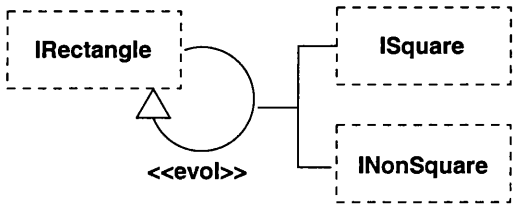

Figure 6.1: Square/Rectangle structure for time efficiency.

The time–efficiency in this example might be small, but there are many situations in which a computationally–intensive result can be obtained much more efficiently for a constrained subset of objects than for the set of objects in general.

6.1.2 Space Efficiency Model

Here, the goal is to avoid storing unnecessary information required by rectangle instances for squares that do not require the additional information (since it is derived from other information plus the constraints for square objects). In this case, we only need to store a single dimension for squares, the constraint indicates that the value of the other dimension is the same — the space saving is small in this example but in other cases the saving may be much greater. The starting point is the same as that for time efficiency: clients do not need to be able to distinguish between square and rectangle instance but appropriate behaviour is required from those instances. This time we must create a variant that uses a different representation at different times.

This can be achieved by having the variant constructed from a rectangle–state abstraction which has two variants: one that stores both dimensions and one that stores only one dimension. When the dimensions of a non-square-state are set to identical values, then a square-state instance is returned (and the non–square–state instance is deleted); the opposite happens when a square–state instance is requested to change its dimensions to non–identical values. This structure is shown in Figure 6.2, ISquare and INonSquare provide state–specific implementations and are variants of IRectState which provides a secondary type through which to access these implementations. The IRectangle variant is then constructed from an IRectState instance which can vary dynamically between an ISquare instance and an INonSquare instance, as required.

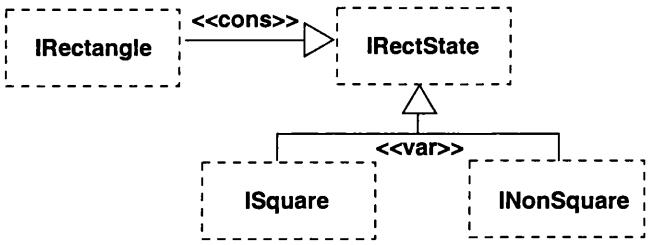

Figure 6.2: Square/Rectangle structure for space efficiency.

Note that the identity of the instances used in the construction of the rectangle variant is not important; it is the identity of the rectangle variant instance that must be, and is, preserved.

The solution outlined above keeps the same instance if possible; this is appropriate if constructing the object is expensive. If not, the rectangle–state abstraction could be made immutable with a new instance returned after every modifying operation.

6.1.3 Restricted Views Model

This time, square and rectangle abstractions have been identified as separate domain abstractions and therefore require separate primary types, squares are mutable and must maintain their constraints. Substitutability is also required for behaviour that will not violate the constraints of square objects. For example, we might have a drawing abstraction that knows how to display rectangles. It

could also display squares without modification. The drawing abstraction only requires read access to square objects so it cannot violate their constraints.

This scenario can be modelled by introducing a new TRORect type, representing read–only rectangles. A rectangle can safely be viewed as a read–only rectangle by introducing a IRORect variant of TRORect which is also a view of TRectangle; similarly, a square can be viewed as a read–only rectangle by introducing an IROSquare variant of TRORect which is also a view of TSquare (Figure 6.3). Under this view, rectangles and squares can be used interchangeably. The drawing abstraction can be written in as a client of the TRORect type (or evolved to use this type). The TSquare and TRectangle types, representing squares and rectangles, are not related, these types can be used by clients that specifically require either a square or a rectangle. However, square and rectangle objects can be viewed in terms of the TRORect type to be used in read–only contexts.

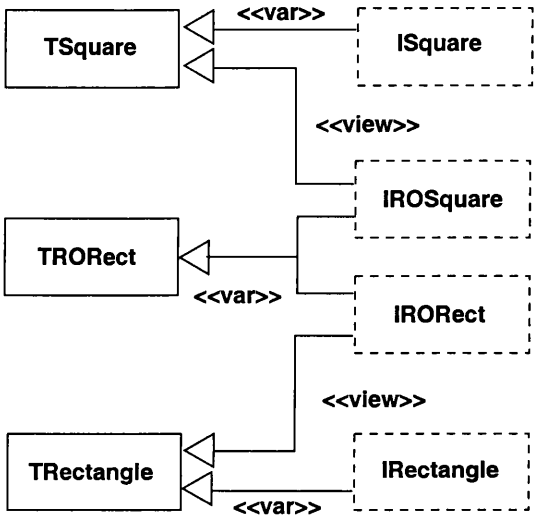

Figure 6.3: Square/Rectangle structure for restricted substitutability.

Note that if the square abstraction is introduced after the rectangle abstraction and we wish to pass square objects to existing rectangle clients then we must evolve those clients to access rectangles via the read–only view (which they must be able to do or we could not guarantee that they could use square instances safely). The clients will then also be able to handle square instances and rectangle instances interchangeably.

This approach is not restricted to read–only views, other views applicable to both kinds of object may be developed. For example, a move operation which moves the bottom left corner of a shape to a new location (moving the rest of the shape relative to it) cannot violate the constraint introduced in the square abstraction. Therefore it would be possible to have a movable shape type via which both rectangles and squares can be accessed.

6.1.4 Full Substitutability Model

Suppose that full substitutability is required such that all rectangle operations can be applied to squares. Additionally, there are special operations that squares offer to their clients that are not

offered by other rectangles so a separate primary type is required for squares.

To achieve full substitutability whilst still offering operations that can violate constraints, we must pass some of the responsibility on to clients of the square abstraction. Although squares and rectangles can be used interchangeably for operations that can potentially violate the constraints of squares, we must ensure that the client has sufficient information to avoid violating square instances.

In this case we can define an abstraction, Configurable Rectangle, which defines a constant property 'fixed-aspect-ratio' which may be set to true or false. (Constant properties are set at construction time and cannot be modified afterwards.) All square instances should set 'fixed–aspect–ratio' to true. Operations such as setDimensions can then be written using this property in their preconditions. In this way it is possible to write methods that are only appropriate when the aspect ratio is fixed and other methods that are only appropriate when the aspect ratio is variable. Clients will then need to ensure that preconditions are met before invoking methods.

It is also possible to use specialisation to create a Square subtype, or indeed a Fixed Proportion Rectangle, or a 2:1RatioRectangle, in each case the class would set an appropriate aspect ratio, and fix it during construction. In such a subclass preconditions that are guaranteed by the class invariant need not be checked by subclass clients and additional methods that should not be accessible to superclass clients may be added. Similarly it is safe to disinherit methods that cannot apply to the current class. Clients that access square instances via a Square subclass will know precisely which operations can be invoked.

Clients that only know that they have a Configurable Rectangle will not have access to additional specialised operations, and operation preconditions will inform them of exactly which operations are allowed and the results that can be guaranteed from them. The resulting structure is shown in Figure 6.4.

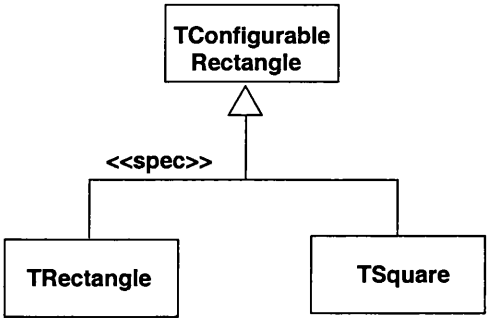

Figure 6.4: Square/Rectangle structure for full substitutability.

6.1.5 Reuse Model

Suppose we wish to maximise reuse between the square and rectangle abstractions in order to minimise future maintenance effort. One option would be to use construction and use a rectangle object to store the dimensions of a square, the square subclass would then only invoke methods that will not violate its additional constraint.

A better approach would be to use idea of a Configurable Rectangle abstraction from the previous section, but this time, at the realization level. We can construct both ISquare and IRectangle implementations from an IConfigurableRectangle implementation, as in Figure 6.5. In the case of the rectangle we do not fix the aspect ratio when creating a corresponding configurable rectangle instance but for a square, we do fix the aspect ratio. This means that if the preconditions associated with configurable rectangle instances, used in the construction of a square, are maintained, then the constraint within the square class will be maintained.

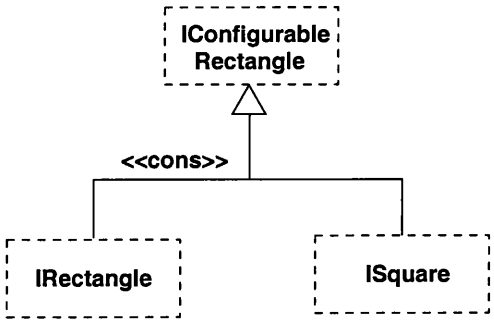

Figure 6.5: Square/Rectangle structure for reuse.

Note, that the reuse relationship does not necessarily correspond with a subtyping relationship between the corresponding types. If the types for square and rectangle are unrelated then the square method will not offer any methods that can violate its constraints and its clients will not need to do precondition checking.

6.2 Binary Methods: Points and Coloured Points

The point/coloured–point problem, as presented in Chapter 2, is a representative example of a class of problems in which subclass values are extensions of superclass values, and in which those extensions need to be considered in interactions between instances of the superclass and subclass. The problematic method, in the example, is the equality method that coloured point inherits from point, it is likely that coloured point will require an equality method that compares coloured points in terms of location and colour. The point equality method should be used when comparing points, the coloured point equality method should be compared when comparing coloured points. It is less clear what should happen when a point is compared with a coloured point, and vice versa.

Methods that rely on two or more arguments (including the receiver) in order to determine which method to invoke are termed binary methods. The problem of typing binary methods has been of the key issues in inheritance research for the past few years. The majority of the work to date has focussed on achieving type safety. Alternative solutions have resulted in different semantics for binary methods [Bruce, Cardelli, Castagna, The Hopkins Object Group, Leavens and Pierce, 1995]. Matching and other forms of parametric polymorphism support the typing of methods that require that the dynamic types of the receiver and argument are identical; multiple dispatch allows methods to be typed precisely (that is for any combination of dynamic types); and a further suggestion is

to combine the arguments into a single object for the purposes of the method invocation and to dispatch on that object.

Here we discuss the conceptual issues behind the potential relationships between point and coloured point classes and show how the SIR framework can support the development of appropriate modelling solutions. A number of scenarios are discussed, some of which require a particular semantics for binary methods and some of which do not involve binary methods at all.

6.2.1 Reuse Model

Perhaps the simplest relationship between points and coloured points is a reuse relationship in which the location part of a coloured point is implemented using construction from a point. This is appropriate if points and coloured points do not need to be related by type. For example, coloured points might be a graphical abstraction whereas points are simply an implementation abstraction, used in the implementation of coloured–points, rectangles and other graphical objects. This arrangement is illustrated in Figure 6.6.

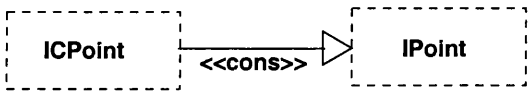

Figure 6.6: Point/ColouredPoint structure for reuse.

The coloured point abstraction might be constructed from a mutable point, or it might be constructed from an immutable point, dynamically changing the superclass instance after every modification. Since there is no type relationship between the subclass and superclass in this case the issue of typing binary methods does not arise.

6.2.2 Evolution Model

Consider a graphical application that was developed for use on black and white (or other two–tone) screens. The point in this application has no colour property — there was no requirement for a colour property since it could not be changed, all points would have been drawn in the current foreground colour. Now consider updating the same application to make use of colour screens. The point abstraction in such a system should obviously be coloured. A non–coloured point has no rôle to play as a graphical abstraction in such a system.

The natural relationship between the old point abstraction and the new (coloured) point abstraction is evolution. A colour property can be added using to the point type using evolution together with additional operations and specification to take the colour property into account. Similarly variant implementations of point must be updated to implement the colour property and its associated new operations, and to extend existing methods as necessary. This approach is shown in Figure 6.7.

In an evolution relationship, it is not possible for instances of the new and old type to coexist, this means that when creating a new version of a binary method it is only necessary for the method to

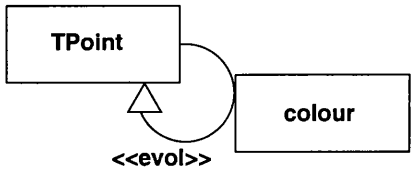

Figure 6.7: Point/ColouredPoint structure for evolution.

apply to instances of the new (coloured) point type (there will be no instances of the old non–coloured point in the resulting system). This can be seen as covariance when both types are compared, but since the new type replaces the old type it can also be viewed as invariance. Evolution does not require any special treatment of binary methods.

6.2.3 Full Substitutability Model

If a coloured–point abstraction is to be fully substitutable with a point abstraction then it must support all of its operations in a conformant manner. This would mean that an equality method, inherited from point, that said it would return 'true' if the x and y coordinates were equal, could not be overridden to accept only coloured point arguments in a coloured point subtype. It is possible to achieve a specialisation relationship between point and coloured point in this situation but it is necessary to consider the real meaning of the equality method.

As far as supertype clients are concerned, equality means that the point is at the same place in 2-D space. An approach to creating a conformance relationship is to acknowledge this and replace the equality method with a more appropriately named same2DLocation method (evolution of the abstraction and of its clients will be required if the point abstraction already exists). The coloured point type will inherit this method and will be able to support it unchanged. The coloured point type can then add a sameColour method which only accepts Coloured Points for comparison.

In the resulting system there is no general way in which to compare points and coloured points for equality. This is the appropriate solution if such comparisons do not make sense in the context of the current system.

6.2.4 Client–Specified Comparison Model

Although there may be no general way of comparing points and coloured points for equality in the previous example, there may be comparisons in certain situations. For example, in a context where colour is not important it may be appropriate to define an equality method which compares only x and y coordinates.

It is the client that decides whether or not colour is important. It is possible to create a view that can apply to both points and coloured points in which colour is ignored and equality is defined in terms of x and y coordinates. If coloured point is a subtype of point (as in the full substitutability example) then a single view can be used for points and coloured points. If the abstractions are not

type–related then a view of each will be required. Since the context of the view specifically ignores colour it will only be used by clients that require that interpretation of colour.

The same approach can be taken to creating a colour–dependent equality method. There may be scenarios in which it makes sense to define equality in terms of location and colour. Such a view may be provided just for coloured points; or a view of points may also be created in which they are given a default colour (say, black) or in which they have an undefined colour that is equal to no other colour (including another undefined colour). In Figure 6.8, the TColourComparable type is introduced to allow comparisons based on colour, views of TPoint and TCPoint types are introduced as variants of TColourComparable. By viewing any combination of points and coloured points in terms of TColourComparable, we are able to compare them in terms of colour.

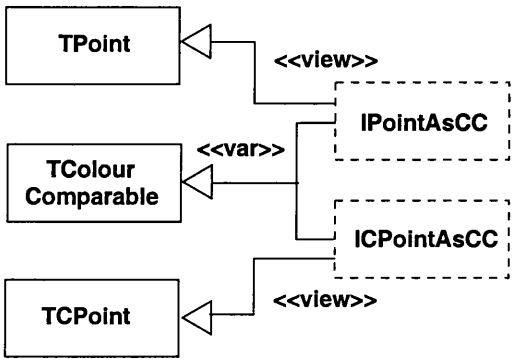

Figure 6.8: Point/ColouredPoint structure for client specified comparisons.

The approach described here allows the definition of equality to be taken outside the main abstraction hierarchy, where it cannot be precisely defined, and into well–defined contexts where a particular notion of equality is required.

6.2.5 Conversion Views Model

In a further scenario, both points and coloured points exist within a system (say for creating black–and–white and colour windows), the types are not related by specialisation, and each has its own equality method. In some situations, for example, editing a coloured item in a black–and–white window and vice–versa, it is necessary to view a point as if it were a coloured point, or a coloured point as if it were a point. In this case both abstractions define their own equality methods and a variant of each can be created as a view of the other. The point view of coloured points ignores colour and the coloured point view of points assumes that the colour is the current default for that window. The required structure is shown in Figure 6.9 where a TPoint instance may be an instance of direct variant IPoint, or an instance of ICPointAsPoint which is also a view of TCPoint. Comparisons made based on the TPoint type will not take colour into account. An analogous approach is taken for variants of TCPoint.

Note that since we have bidirectional views here we should ensure that viewing a point, that is actually a view of a coloured point, as a coloured point, will result in the original coloured point

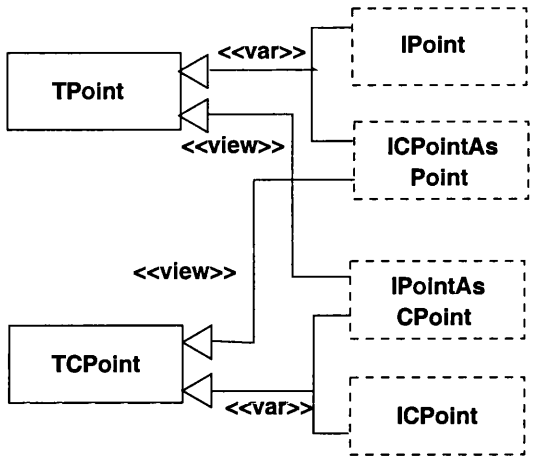

Figure 6.9: Point/ColouredPoint structure for bidirectional views.

and not in a point with a default colour. This is important if we have simultaneous colour and black–and–white editing views of the same abstraction (for example, to see what they will look like when printed on black–and–white and colour printers) which allow objects to be cut from one view and then pasted into the other. View constructors must be able handle this case (potentially via a special language construct).

6.2.6 Matching Model

Consider an algorithm for updating a matrix display where updating is an expensive operation. Lists of points to add and remove are provided but we only want to draw a point if it is not identical to the one currently being displayed. Both monochrome and colour displays are available. The algorithm for both should be the same except for the interpretation of 'identical'. Two points on a monochrome display are identical if they have the same location; for a colour display the points must also have the same colour. Cross–comparisons are considered an error, the system should never compare a point and a coloured point for equality. We do not want to mix comparisons across the point and coloured point types but we do want to use the same client code to handle both. We assume that for other operations, such as setting the location, points and coloured points are interchangeable.

Neither genericity or subtyping alone can handle this situation. The matching relationship described in Chapter 2 (or similar) is required to ensure that two points can only be compared if they have the same dynamic type. Within the SIR model, matching is modelled using specialisation — it is possible to create specialisation relationships in which type is one of the elements that can be conformantly refined in a subtype.

In order to support both a subtyping and a matching relationship between point and coloured point, two types must be introduced for each, one to be used in order to get subtyping and one to be used to achieve matching. Any operation that can be used in a subtyping context can also be used in a matching context so the matching type must be a subtype of subtyping type (via an abstraction relationship). The resulting structure is shown in Figure 6.10. The matrix display code

will be written in terms of the MPoint type which will allow same–type comparisons, but not mixed comparisons. Other parts of the system, such as logging, can be written in terms of the TPoint type, TPoint allows points and coloured points to be used interchangeably, but does not support equality tests.

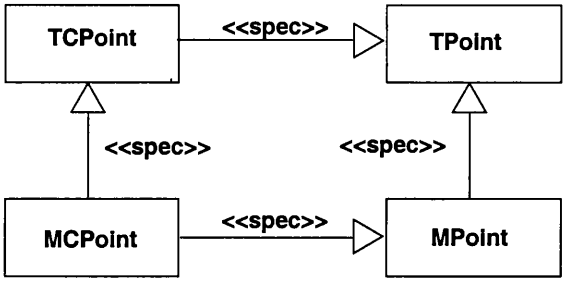

Figure 6.10: Point/ColouredPoint structure for subtyping and matching.

Of course, the matching relationship can only be made use of if the implementation programming language supports it.

6.2.7 Multimethods Model

Now consider an environment where the type of both arguments must be considered in order to correctly resolve a comparison. For example, two points are equal if they have the same location, two coloured points are equal if they have the same location and colour, and a point and a coloured point can be compared but are never equal. In other circumstances, coloured points must behave as subtypes of point.

This solution requires multiple dispatch which is supported by the SIR model, as discussed in Chapter 4. Using multimethods also requires the supertype to know about the subtype. This is generally considered a bad thing, but in this case it is appropriate since subtype and supertype are conceptually strongly coupled. If the point supertype exists before its coloured point subtype then the creation of the subtype, and the close conceptual relationship between the subtype and supertype requires the evolution of the supertype's behaviour in accordance with its new environment. The multimethod approach to the point/coloured–point relationship is shown, in pseudo code, in Figure 6.11.

```
interface TPoint
{
   multimethod equals(TPoint);
   multimethod equals(TColourPoint);
}

interface TColourPoint specialises TPoint
{
   multimethod equals(TColourPoint);
   multimethod equals(TPoint);
}
```

Figure 6.11: Multimethod equality.

6.3 Case Study: Web Site Manager

In this case study we model the static perspective of a system in its entirety to illustrate how the SIR framework can be used for system design. The chosen case study is a web site manager. Web sites often contain a number of 'projects' each of which contains many web pages with a consistent 'look and feel'. In the past, the consistency has been managed manually, obviously this is both time–consuming and error prone, an improved approach is to identify elements that are shared across a site and allow these to be modified in a single place with changes applying across the site.

System Requirements The web site manager must be able to display and support the editing of two kinds of configuration file:

1. **informational config files** — allow variable aspects of the system to be specified, for example, the document author. Informational configuration files are organised hierarchically so that they can be overridden at various levels within a project.
2. **structural config files** — describe a particular document subdivision, such as a header, footer, menu or advertisement that appears in multiple documents throughout a site. Structural config files rely on informational config files for variable aspects of their content, such as obtaining the author of the current document for inclusion in a footer.

It should be possible to view each configuration file in a way that allows it to be modified easily — in simple cases this may mean that the file can be directly edited, in other cases a more powerful interface may be required.

Non–Functional Requirements The resulting system must run within a web browser.

Components that are reusable outside the current context should be developed in preference to system–specific ones.

The system should be implemented using the Perl 5 language to fit with existing related tools. The Perl 5 language provides mechanisms for implementing systems in both object–oriented and non–object–oriented styles. There is already a large repository of modules written in Perl 5 and the system should make use of such modules where appropriate.

6.3.1 Design of the Web Site Manager

We use a design in which a project editor abstraction has overall control for the application, and the editing of configuration files is handled by specialised editors. The project editor must be able to offer the user a choice of configuration files and invoke specialised editors for selected configuration files.

Project Editor

The Project Editor part of the system is responsible for managing the editing of the configuration files via the web interface. The project editor must be able to present a list of the configuration files associated with a particular project and it must be able to create valid web pages incorporating the editor for a particular configuration file. The Project Editor itself should have overall control for creating web pages so that it can add further elements to a page as appropriate, for example it may add a menu to allow a different configuration file to be selected for editing. The Project Editor will insert the editor for a particular configuration file at the appropriate point in the web page.

The Configuration File

A configuration file is responsible for maintaining information regarding a particular structural element in a web project. There are a number of different kinds of configuration file from the very simple (read from file, edited and written out to file) to the complex (such as menus which require custom readers, editors and writers). Each configuration file will contain a number of Elements (which may be HTML code or scripting code) that will need to be edited. There is no important distinction to be made between configuration files that have informational content and those that have structural content; they can be treated identically for design purposes. Clearly the methods used to implement the various kinds of configuration file will be different. However, the Project Editor will also need to be able to access the various configuration file objects polymorphically.

Normally, this analysis would suggest a subtyping inheritance relationship (specialisation in the SIR model). However, further analysis reveals that we do not just want to be able to treat the objects as polymorphically equivalent, we also want their full behaviour to be accessible from the same clients — we do not want to have to write a specialised Project Editor every time a new configuration file subtype is added. This is essential if developers are to be able to develop new kinds of configuration file and simply plug them in to the system.

For this reason we do not introduce a new type for each kind of configuration file, we model them as variants of a configuration file base type which they all implement appropriately. Figure 6.12 illustrates this situation.

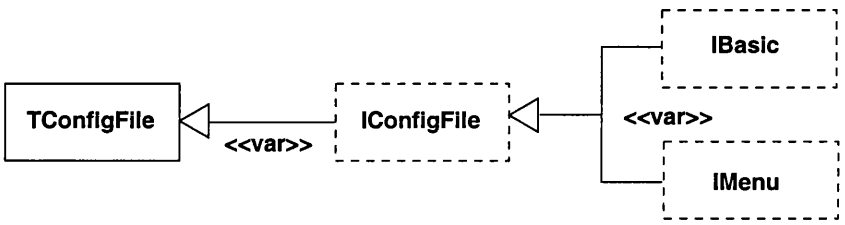

Figure 6.12: A configuration file type with multiple variants.

The use of variants provides a number of advantages in the rest of the system:

1. We can ignore the configuration file variants when we are not designing them directly. The remainder of the system can be designed in terms of the TConfigFile type. This will result in

a simpler system than having each variant lead to a separate type requiring specialised clients for some aspects of behaviour.

2. We can design the remainder of the system without considering individual variants further. This means that we can implement the Project Editor without designing the variants in detail, perhaps providing a single variant for testing.

3. We can introduce additional variants once the system has been developed in order to handle new user requirements. We can be certain that no other part of the system will require modification when a new variant is introduced.

Designing the Implementation of the Configuration File Variants

In this example we have the foreknowledge that the variants that we are developing are only a subset of all possible variants. It is therefore important that we design with reusability in mind. We will consider a number of areas where the SIR model can help to focus on reusability.

Separation of Web Interface Although we have no forewarning of this, it is also possible that the web interface to the system is only one potential interface, it is therefore valuable to separate out the aspects of the implementation that are dependent on there being a web interface from those that are not. The web dependent parts of the configuration file abstraction should be part of a view since they are only relevant when the controlling part of the application wishes to use them, it is only the displaying/editing part of the system that is actually dependent on the web interface. The task of reading a configuration file from disk and writing it out again should not be dependent on the web interface.

This observation enables us to restrict the responsibilities of configuration file variants to file reading and writing, and representation of configuration file elements. Views will handle the displaying and editing of the configuration files in a web environment.

Basic File Read/Write The simplest kind of configuration file variant will only distinguish one element: the contents of the file it represents. The variant must offer read and write operations on the file and a representation of the information contained in the file.

Clearly this task is not specific in any way to the system under development except for the fact that it is a configuration file that we are dealing with. This situation suggests that we should create an IUpdateableFile construction unit with the required behaviour (or ideally, use one that is already in existence) and use construction to utilise this behaviour in the IConfigFile variant we are realizing. This results in the relationship shown in Figure 6.13.

There is a Perl abstraction that already has much of this functionality: IO::File (that is, the File abstraction within the IO namespace). It handles the file reading and writing, but does not provide a representation of the current information in the file which can be used and updated by the application. The TUpdateableFile abstraction required in this system should be in a specialisation

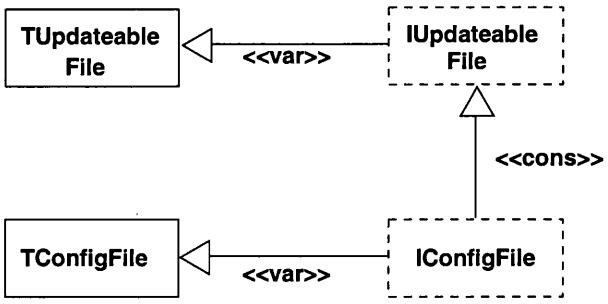

Figure 6.13: Using construction for reusability.

relationship with the existing IO::File abstraction, it should also be in a construction relationship with the implementation of IO::File. IO::File combines both interface and implementation into a single module, as is usual in Perl. We could split IO::File into IO::TFile and IO::IFile for the purposes of modelling, or we can simply use specialisation and construction from the same abstraction, as shown in Figure 6.14. Of course, implementation is not inherited in the specialisation relationship.

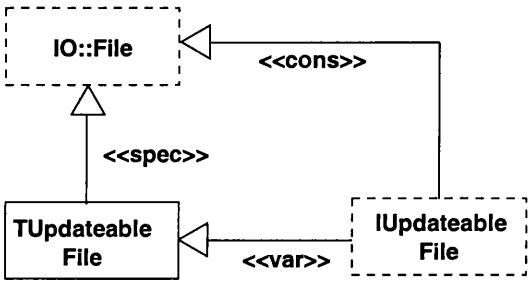

Figure 6.14: Reusing the existing IO::File Perl module.

This approach has enabled us to reuse an existing abstraction IO::File and provide another abstraction TUpdateableFile for reuse in other systems. Note that the specialisation relationship between IO::File and TUpdateableFile is appropriate since the updateable file abstraction is a specialisation of the file abstraction, it performs the same operations but it also contains a representation of the information stored in the file that can be modified by the application and then written back to file. This means that it is possible to substitute a TUpdateableFile whenever an IO::File is expected, thereby creating the potential for reuse of existing client code written for IO::File.

In fact TConfigFile is a valid subtype of IO::File via TUpdateableFile. Introducing such a relationship would allow us to reuse existing client abstractions, it would also mean that all variants of TConfigFile would need to implement the interface of IO::File. This is not too much of a burden since they can all achieve this via a construction relationship with IO::File. On balance the advantages of this approach outweigh the disadvantages so the relationship can be introduced as shown in Figure 6.15.

Web Elements It must be possible to edit each kind of configuration file. This functionality is not provided by the TConfigFile abstraction itself, as explained above. It is appropriate to use a

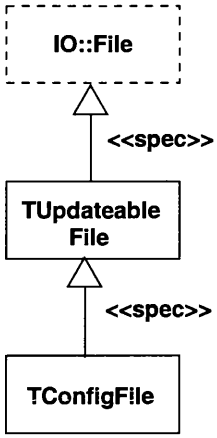

Figure 6.15: A specialisation relationship with IO::File increases reusability.

view of TConfigFile that creates the required HTML code from the information stored in instances of TConfigFile and updates this information after editing so that it can be written back to the file. Since the details of the editor will potentially be different for each variant of TConfigFile, multiple view implementations will be required — one for each variant or sub-hierarchy of variants which differ significantly.

The communication between the Project Editor and the editor views is minimal: it must simply request the HTML code to be displayed. For this reason we introduce a TWebElement type of which the editor views are variants. The TWebElement type can then be reused in other scenarios that require objects that produce a portion of displayable HTML code. This situation is illustrated in Figure 6.16.

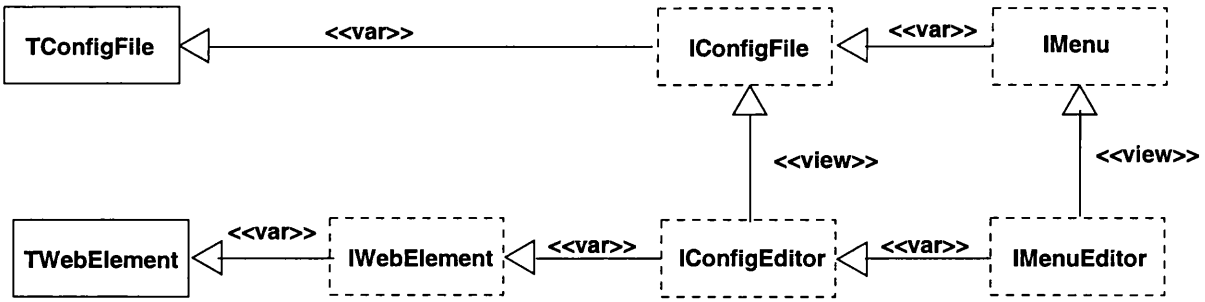

Figure 6.16: Views of TConfigFile as variants of TWebElement.

TWebElement is reusable, in fact, it can be reused within this system to handle non–editable web page elements such as headings, footers and documentation to be used in the project manager system itself.

Note that if we also view the configuration files in terms of an interface that is not file–specific, then we could replace the back–end of the system with one that stores configuration details in a database.

CGI Functionality Each web editor must be able to display the various components of its editing view of a configuration file in a browser. This functionality is provided by existing Perl modules:

CGI::Request and its subclass CGI::Form [Bunce and Stein, 1998] which additionally provides an interface for form generation.

For generality, the IConfigEditor abstraction should be a variant of an IWebEditor abstraction that supports form–based editing. The IWebEditor realization does not need to inherit from the CGI::Form module. The characteristics of the CGI::Form module are concerned with the provision of methods to generate HTML, the CGI controller on the other hand is concerned with generating HTML. It is sufficient to use clientship here. This is an important distinction between construction and clientship. It would only be appropriate to use construction if the HTML generation methods properly belonged to the web editor abstraction; this is clearly not the case.

6.3.2 Adding Revision Control

We now consider a requested change to the system: the addition of revision control so that previous versions of configuration files are saved and can be retrieved if necessary.

This will mean evolving the TConfigFile type so that it also offers operations such as checkIn and revertToPrevious. An alternative would be to provide a revision control view of TConfigFile however in this case we want to make revision control functionality an intrinsic feature of the configuration file abstraction so evolution is appropriate. Evolution of the TConfigFile type will obviously require evolution of the variants implementing TConfigFile.

There is an existing Perl model, RCS, that provides revision control using the RCS system, offering functions to check in and check out a file, and so on. We could choose to evolve each variant so that they all inherit from the RCS module, but this would be missing out on an opportunity for future reuse. Revision control is relevant to all files, not just those used as configuration files. We therefore create a subtype of IO::File, TRCSFile the implementation of which is constructed from the implementations of both IO::File and RCS.

We must now decide whether to create a TRCSUpdateableFile type which combines the behaviour of both TRCSFile and TUpdateableFile, or, we must evolve the TUpdateableFile abstraction so that it supports revision control. We choose the latter since we are evolving the notion of an updateable file as required by this system. Note that in environments where RCS is not available the RCS adaption will not be applied to the TUpdateableFile abstractions (and similarly for variant implementations). After evolution, this results in the type hierarchy shown in Figure 6.17.

The evolution of TUpdateableFile therefore takes it from being a specialisation of IO::File to being a specialisation of, IO::File subtype, TRCSFile. This is a valid evolution step since the resulting TUpdateableFile remains usable by all existing clients.

We also need to evolve the variant implementation of the IUpdateableFile abstraction so that it is in a construction relationship with IRCSFile rather than IO::File. In order to introduce RCS capability into the user interface we must evolve the TWebElement views of TConfigFile. Note that this behaviour is not variant specific so we can introduce it in IWebConfig so that it is used by all variants.

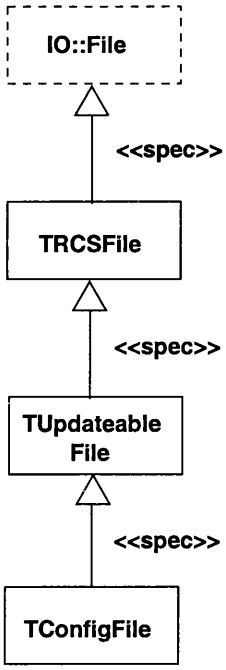

Figure 6.17: The file hierarchy after the introduction of revision control.

6.4 Conclusion

The first two case studies were concerned with determining the possible relationships that could result from two well-known modelling problems through the analysis of specific examples. In both cases the SIR framework provided a structured basis for conceptual analysis of these problems. The analyses took into account the possible contexts within which the relationships could arise and provided modelling solutions for each context. The availability of the SIRs enabled the solutions to be described at a high-level, rather than in terms of programming language mechanisms. In addition to illustrating the power of the SIR framework, the resulting solution sets are valuable in their own right, supporting the selection of the correct conceptual solution in a particular context.

The final case study in this chapter developed the design of a system for editing web site configuration files. Even though the system was small, all five SIRs were used in the system design in a natural manner. Perl 5 was chosen as the target programming language partly because it provides a large set of freely available modules that could be reused. Although the target implementation language is dynamically typed and generally very flexible, using the SIR approach for designing a system provides structure and aids understandability (and therefore maintainability and reusability). The SIR framework provided a structured environment in which existing modules could be reused, and new, reusable modules could be created. The richness of the relatively simple system in this case study illustrates that there is an overwhelming need to have a means of modelling the conceptual complexity that is inherent even in small systems — the SIR model provides just such a means.

The three case studies in this chapter provide further illustration of how the five SIR relationships provide a sufficient replacement for current inheritance mechanisms, and in fact exceed the conceptual modelling power and clarity currently available in the inheritance mechanisms of mainstream

object–oriented programming languages.

Chapter 7

Implementation Techniques

The SIR framework is concerned with developing structured designs. A system designed within the SIR framework could be implemented in any programming language, but the transition will be smoothest if the target programming language supports the design constructs directly. If the target language does not offer direct support, then in order to achieve the correct semantics it may be necessary to introduce extra classes and objects that are not present in the SIR design for a particular system, and in some cases there may be no appropriate implementation solution.

In order to achieve maximum language–level support for the SIR framework it would be necessary to develop a new programming language which is beyond the scope of the present work. Instead we consider the Java language as a basis for the implementation of SIRs. First, we consider the current level of support for the SIRs in the Java language, and then we consider how Java could be extended to provide full support for each SIR using proposed extensions to the Java language and meta–object protocol systems for Java to provide additional flexibility. The intention is to illustrate ways in which the SIRs could be supported rather than to provide a direct implementation solution.

Language–level support is not the only possible approach to providing a seamless transition from design to implementation, another approach is to add support at the CASE tool level. If a development environment provides direct support for the SIR framework, complete with automatic code generation, then it is not necessary to have constructs that map precisely to the SIR constructs. It would be sufficient to have patterns that the CASE tool can use to map from design–level constructs to programming language constructs. Some techniques that would be too complex to be hand-coded could still be of use in an automatic code generation environment.

7.1 Levels of Support

There are different levels at which a language can be said to support a relationship. This is noted by Stroustrup in [Stroustrup, 1988]:

> A language *supports* a programming style if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case the language merely *enables* programmers to use the technique.

When considering how the SIRs can be represented in Java, we are aiming for support, rather than simply enabling programmers to implement systems that have been designed within the SIR framework. It is useful to distinguish between three dimensions of support for SIRs in a programming language:

1.
   - **Direct Support** — There is a mechanism in the language that corresponds directly to the relationship and which supports or enables the implementation of some or all aspects of the relationship (the exact nature of the construct may be influenced in its detail by the nature of the implementation language).
   - **Indirect Support** — The relationship (or some aspects of it) can be implemented in the language but multiple constructs must be used in combination to achieve the appropriate semantics.
2.
   - **Complete Support** — All aspects of a relationship can be implemented in the language.
   - **Partial Support** — Certain aspects of a relationship can be implemented in the language but others are not supported. It is possible to implement a subset of SIR models within the language.
3.
   - **Strong Support** — The correct semantics can be achieved and the constraints imposed by the relationship will be enforced.
   - **Weak Support** — The correct semantics can be achieved but constraints imposed by the relationship must be manually enforced.

A language construct fully implementing an SIR would provide direct, complete and strong support which we will refer to as **full support**.

7.2 Implementation Approaches

When an SIR cannot be fully implemented using an appropriate construct in a particular language there are a number of approaches to adding support for the relationship without modifying the existing language:

Implementation Pattern If a language offers indirect support for an SIR then an implementation pattern can be used to capture the technique. This allows the relationship to be represented consistently across systems implemented in that language.

For example, in Java there is no support for assertions. An implementation pattern for preconditions is to introduce a conditional statement at the beginning of a method which generates an error if the

precondition does not hold. A debug variable can be used to turn precondition checking on and off. This implementation pattern is inferior to a built–in assertion mechanism but can achieve limited assertion functionality within the programming language.

Compile–Time Meta–Object Protocol If a relationship cannot be supported in a simple way within a language but can be simulated using complex techniques then it may be possible to use a compile–time meta–object protocol (MOP) for that language. A compile–time MOP supports transformations on a program to provide support for techniques such as the extension of a language by adding new constructs which the MOP then maps to an implementation in the target language. For example, a compile–time MOP could be used to introduce a new keyword.

The C and C++ preprocessors can be seen as a simple form of MOP: before templates were added to C++ as a language construct, they could be implemented using preprocessor macros. More advanced compile–time MOPs such as OpenJava [Tatsubori, 1998] provide preprocessors in which a program can be directly manipulated in terms of its object–oriented structure.

Run–Time Meta–Object Protocol In some cases compile–time code transformations are not adequate to support a particular behaviour. In this case it may be possible to use a run–time MOP which provides reflexive access to a system at run–time. The exact features available at run–time vary between MOPs.

For example, limited reflexive capabilities are available for the Java language, these are a standard part of Java but exist outside the core Java language. For example, it is possible for a CASE tool to use the reflexive capabilities of Java to determine the type of an object and the methods applicable to it — these activities are not possible within the Java language itself. Guaraná [Oliva, 1999] is a powerful run–time MOP for Java.

7.3 Support for Specialisation

Specialisation introduces a subtyping relationship between two types and leads to behavioural, as well as syntactic, substitutability. We consider current support for specialisation in Java and then look at how full support could be achieved by extending the Java language.

7.3.1 Current Support for Specialisation in Java

Java provides better support for specialisation than other object–oriented languages because it provides an interface construct that allows types to be introduced without a corresponding implementation. Interfaces are intended to be used in cases where several different objects need to implement the same protocol. Within the SIR model we would introduce a view in many such cases to allow the instances of different classes to be used interchangeably for a particular set of behaviours. When representing the SIR model in Java, the interface construct can be used to introduce primary types.

The extends relationship between Java interfaces corresponds to specialisation (see Figure 7.1) since it provides substitutability at a syntactic level. The extends relationship between classes also corresponds to (but is not limited to) a specialisation relationship since classes in Java also introduce types.

```
interface UniversityMember
{
    String name();
}

interface Staff extends UniversityMember
{
    String staffNumber();
}
```

Figure 7.1: Specialisation using extends in Java.

Java's support is weaker for other aspects of specialisation. Java does not provide any mechanism for introducing behavioural specifications so support for specialisation is at a purely syntactic level.

Additionally, Java permits the static overloading of methods based on the type of arguments. This is not supported within the SIR model of inheritance since it leads to confusing behaviour in the presence of subtyping: the same method invocation, on the same object, with the same argument(s), can result in different behaviour based on the static type of the argument(s).

The SIR model supports multimethods and matching, although these advanced techniques can be omitted from the model to provide a simpler subset. Java does not support either multimethods or matching.

7.3.2 Extending Java for Specialisation Support

As we have seen, Java offers weak support for specialisation by using interfaces for primary types and the extends relationship between them to model specialisation. Since the extends relationship in Java is also used to relate classes, full support for specialisation would require the introduction of a separate specialisation relationship between interfaces only, for example, using a specialises keyword as shown in Figure 7.2.

```
interface Customer
{

}

interface PrivateCustomer specialises Customer
{

}

interface CorporateCustomer specialises Customer
{

}
```

Figure 7.2: Specialisation extension to Java.

In order to provide complete support for specialisation Java would also need to support assertions for

the formal specification of interfaces with design–by–contract style rules for inheritance of assertions. Although Java itself does not provide support for design–by–contract with assertions, there are language extensions that add specification capability to Java, such extensions include Larch/Java [Ciancarini and Cimato, 1997] and iContract [Kramer, 1998]. A simple Java interface including an iContract specification (adapted from an example in [Kramer, 1998]) is shown in Figure 7.3, the assertions that appear in comments are handled by the iContract Java preprocessor which converts them into Java code. The assertions are included in comment tags so that they will be ignored by Java compilers — this means that the resulting program is still valid Java.

```
interface Person
{
  /**
   * @post return > 0
   */
  int getAge();

  /**
   * @pre age > 0
   */
  void setAge(int age);
}
```

Figure 7.3: An iContract specification.

Using interfaces to correspond to primary types in Java and formally specifying abstractions using Larch/Java, iContract, or similar, provides a high level of support for the specialisation relationship.

The SIR framework also supports matching and multiple dispatch and neither of these advanced techniques is currently supported by Java. Additionally, Java supports method overloading which is not allowed by specialisation and would need to be replaced by multiple dispatch semantics.

The Pizza [Odersky and Wadler, 1997] extension to Java provides support for the implementation of matching, Pizza supports the creation of classes that are parameterised by the self type (the type being defined). The term F–bounded polymorphism is used to refer to the matching relationship.

The Kiev [Kizub, 1998] extension to Java supports multiple dispatch for multimethods and could be used to implement this aspect of SIR specialisation. Kiev's multimethods allow the modem connection problem of Chapter 2 to be implemented as shown in Figure 7.4.

```
class Modem
{
   multimethod connect(Modem m); // generic connection method
}

class XModem inherits Modem
{
   multimethod connect(XModem x); // XModem-specific connection method
}

Modem x1 = new XModem();
Modem x2 = new XModem();

x1.connect(x2);      // invokes XModem-specific connection method
```

Figure 7.4: Multimethods in Kiev.

The only problem with multimethod support in Kiev is that it uses method declaration order to resolve any ambiguity. Ambiguity occurs when there is no most specific method for a particular combination of arguments, different methods are most specific for different argument types. Ambiguity is not allowed within the SIR model and such cases must be disambiguated by the introduction of a more specialised method which applies in the ambiguous case. An SIR–aware CASE tool could be used to ensure that ambiguity is removed at the design stage. Note that Kiev also offers Pizza–style parametric–polymorphism so it could be used to add multimethods and matching to Java.

In order to ensure that an SIR model does not use method overloading based on different static argument types, it would be possible to implement a Java preprocessor, using a compile–time MOP, that would flag method overloadings as compile–time errors. If multimethod support was available, for example via Kiev, then overloading syntax could be interpreted as a multimethod, as it should be in the SIR model, and converted to the appropriate multimethod syntax.

7.4 Support for Variant

The variant relationship supports multiple implementations of the same abstraction which may be a type or a partially implemented class. We discuss the level of support for the variant relationship found in the Java language, and then consider how this support could be improved through language extensions.

7.4.1 Current Support for Variant in Java

Weak support for inheritance from a single supervariant is not too much of a problem, primary types can be implemented as interfaces, as discussed above, and implementation classes can be implemented using classes. There are two relationships in Java that correspond to variant, the `implements` relationship must be used when the supervariant is an interface whereas the `extends` relationship must be used when the supervariant is a class.

The variant relationship can be supported in Java by programming according to the maxim that classes do not introduce new primary types. This approach would require the introduction of an interface corresponding to any new abstraction that needed to be used as a primary type. In this way, we ensure that it is always possible to introduce a new variant of an existing type — by simply implementing the interface that represents that type. When a variant relationship is between a partial implementation and a more complete one, the `extends` relationship must be used rather than the `implements` relationship. The major problem with this approach is that Java supports only single inheritance of implementation so it it not possible for a subvariant to have multiple supervariants.

Anonymous classes in Java also provide support for the variant relationship. Anonymous classes allow the creation of individual objects conforming to a particular interface. In other words, one–off variants can be created. This is a convenient way of specifying variants that are intended to have only a single instance. Anonymous classes are especially appropriate to the variant relationship

since they have no name (corresponding to the secondary type they introduce) and must therefore be referred to by the name of the interface (primary type) which they implement. This technique is not required to support the variant relationship but it is a useful shorthand. An example anonymous class (from [Sun Microsystems Inc., 1997] with comment added) is shown in Figure 7.5, an instance of a nameless subclass of Enumeration is defined and instantiated.

```
Enumeration myEnumerate(final Object array[]) {
        return new Enumeration()
        // Anonymous variant of Enumeration
        {
            int count = 0;
            public boolean hasMoreElements()
                { return count < array.length; }
            public Object nextElement()
                { return array[count++]; }
        };
    }
```

Figure 7.5: Anonymous classes implement variants in Java.

The SIR model also requires that primary types only refer to other primary types — it is not possible to introduce a dependency from a primary type to a secondary (implementation) type. This constraint cannot be introduced in Java, but an implementation pattern can be used. Java has the notion of a package, a set of closely related interfaces and classes in which the classes have privileged access to each other. One approach to the separation of primary and secondary types would be to make interfaces (primary types) and variant constructors public while giving package-only visibility to any additional methods introduced by variants. Discipline would still be required to avoid using variants as types outside of the package, but there would be no reason to use variants in this way since they would offer no functionality over that offered by the primary type. This approach would provide opportunities for code–reuse with high coupling within a package (based on variant–specific methods) but would enforce weak coupling between packages. Note that inheritance can still be used across package boundaries but the superclass will always be a primary type rather than a secondary type.

Finally, the SIR model allows subclasses to define extensions to superclass methods which are executed in addition to the superclass method. Java enables method extensions to be implemented by invoking the special **super** method at the appropriate point in a method body. The **super** method will invoke the superclass version of the current method.

7.4.2 Extending Java for Variant Support

The variant relationship can be implemented in Java but Java does not offer full support. For full support we would expect programming language terminology to reflect that of the SIR model by using a single `variantOf` relationship, as shown in Figure 7.6, instead of having all `implements`, and some `extends` relationships corresponding to variant.

Full support for variant would also require method extensions such as before and after methods. Before and after methods can be introduced using a compile–time MOP — methods are rewritten

```
interface TNetworkService
{

}

abstract class INetworkService variantOf TNetworkService
{
   // Base functionality for network services
}

class IPrintService variantOf INetworkService
{
   // Specialised printing capabilities
}
```

Figure 7.6: A `variantOf` relationship needs to be added to Java.

to ensure that the correct extensions will be invoked at run–time.

Additional support is required to ensure that the constraints of the SIR model are enforced: primary types must not refer to secondary types. Additional compile–time checking, implemented using a compile–time MOP, could be used to reject programs, such as the one in Figure 7.7, in which primary types refer to secondary types. In Figure 7.7, a primary type representing ordnance survey maps has a method which returns the locations of monuments on the map, this method has a secondary type, SparseMatrix, as its return type which is not permitted. This program should be rejected at compile–time.

```
interface Matrix
{

}

class SparseMatrix implements Matrix
{

}

interface OrdnanceSurveyMap
{
  SparseMatrix monumentLocations();
}
```

Figure 7.7: Invalid use of a variant.

Additionally, Java does not offer support for multiple supervariants which are permitted within the SIR model. An approach to providing support for multiple inheritance of implementation would be to linearize a multiple inheritance hierarchy. An example linearization is shown in Figure 7.8, a second subclass (B in the diagram) is transformed into a subclass of the first superclass (A) the subclass (C) then inherits from this class. The implementation provided in the second superclass (B) would need to be copied to the intermediate class so that the subclass (C) inherits all of the required implementation. This program transformation could be achieved using a compile–time meta–object protocol.

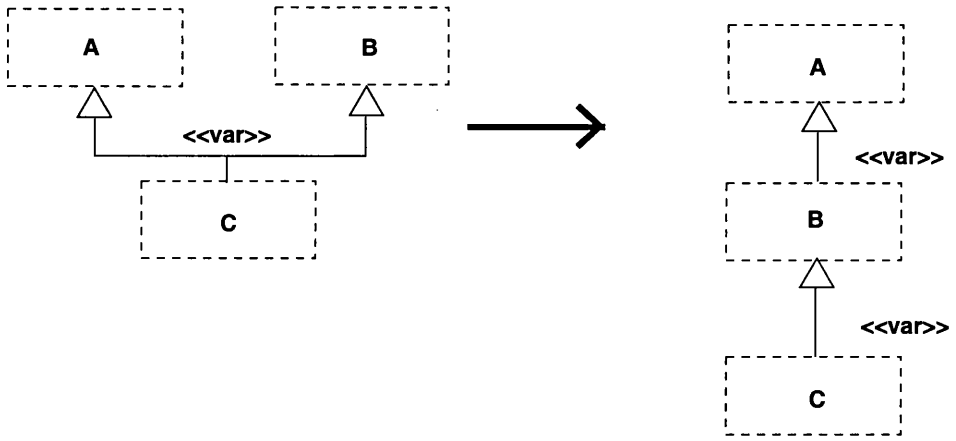

Figure 7.8: Linearization of multiple supervariants.

7.5 Support for Construction

Construction is a relationship specified at the class level which allows an instance of one class to defer a part of its implementation to an instance of another class. We consider the current, indirect support for construction in Java and then consider extensions that would offer a higher degree of support.

7.5.1 Current Support for Construction in Java

One way of representing construction is for the subclass in a construction relationship to use explicit forwarding to an instance of the superclass. This approach cannot be said to support SIR construction since it is time–consuming and error prone — it can only be said to enable construction to be implemented. As with other SIR relationships support for specification is required to fully support construction: when reusing methods it is necessary to fully understand their behaviour. Techniques for adding specification support to Java were discussed under specialisation but also apply to other SIRs including construction.

7.5.2 Extending Java for Construction Support

Support for construction in Java is weak, but the Jamie extension to Java [The Jamie Developers, 1998] supports the type–safe forwarding of a set of methods (specified as an interface) from one instance to another as shown in Figure 7.9 using one of the standard Jamie examples. Control over the access level of forwarded methods is also possible so forwarded methods may be made publicly accessible but need not be.

If multiple objects that support the same operation are delegated to in Jamie then the ambiguity must be resolved by implementing the operation on the delegating class. The SIR model requires that multiple versions of an operation in a construction relationship will all be invoked unless there is an operation on the delegating class, in which case that will be invoked.

```
public class Delegate forwards Target to target {

  private Target target;

  Delegate(Target aTarget) {
    target = aTarget;
  }
}
```

Figure 7.9: Example of forwarding in Jamie.

Support for complex construction, involving multiple objects supporting the same method, would need to be added to Java to fully support construction. This behaviour could be layered on top of Java/Jamie using a run–time meta–object protocol such as Guaraná [Oliva, 1999]. Guaraná allows operation invocations to be intercepted and handled by meta–objects at run–time. Such an approach could be used to implement complex construction behaviour by inserting a method in the subclass which is intercepted by a meta–object which then invokes the appropriate method(s).

7.6 Support for View

The view relationship allows objects to offer different interfaces to different clients. We consider implementation patterns for view in Java and then consider how meta–object protocol techniques could offer more direct support.

7.6.1 Current Support for View in Java

Multiple inheritance can be used to implement a restricted form of view as shown in Figure 7.10. A subtype inherits from supertypes corresponding to each of the views that can be applied to it.

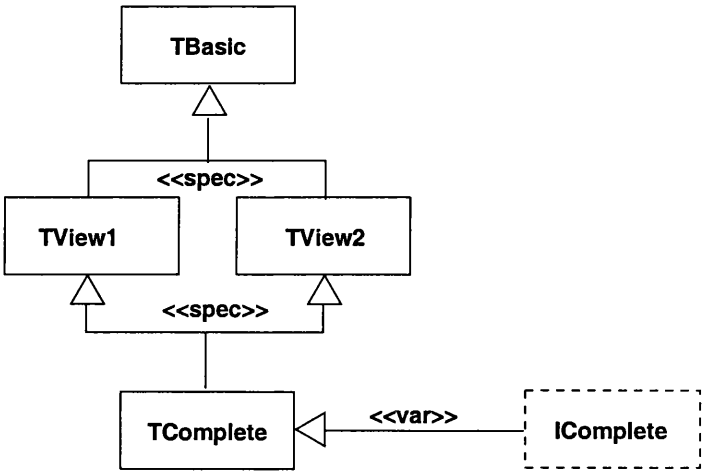

Figure 7.10: Views implemented in a single class.

Per–view state information cannot be maintained using this approach since all view–related state must be stored within the target. Additionally, multiple inheritance forces the creation of a type that combines all views of an object; the SIR model would not introduce this type since it does not

represent a useful system concept. A further problem with this approach is that it does not allow an object to respond in different ways to the same message when accessed via different views. The object must have the same behaviour for a particular operation for every view.

Since Java does not support multiple implementation inheritance, all implementation must be in a common subclass representing the conglomeration of all views. This makes it difficult to modify individual views and easy to introduce dependencies between views that should be independent. This approach does not support the separation of the implementation of views as is required by the SIR model.

Multiple inheritance only provides partial support for the view notion of identity: it does preserve the identity of an object across views but it is not possible to distinguish between different views of the same object. Since views with local state are not possible, it is not meaningful to distinguish between multiple instances of the same of a given object. Multiple inheritance provides a restricted but usable approximation to the view SIR.

Another approach to implementing views in languages that do not provide direct support is to have a separate view abstraction which explicitly invokes operations on a target object (via clientship) as shown in Figure 7.11.

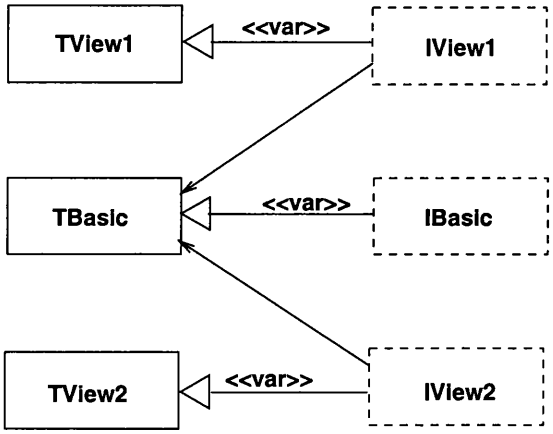

Figure 7.11: Views implemented as separate objects.

This approach does not support the extension of target operations within the view, and it does not preserve identity although the identity of the target object could be made available through the view. Additionally, using an association and explicit method invocation does not provide the conceptual relationship associated with view. It is not possible to distinguish views from any other clients of an object if this approach is used.

In Java, a further approach is possible using inner classes. Inner classes allow the creation of objects that are scoped inside other objects. An inner class can therefore present an alternate view of a class in terms of another type, an example is given in Figure 7.12. Multiple views may be supported in this manner. An inner object has two identities, its own unique identity and the identity of its outer object. This meets the SIR requirement that different views of the same object have the same underlying identity.

```
interface Pair
{

}

interface Point
{

}

class IPair
      implements Pair
{
   class IPairAsPoint
         implementsPoint
   {
      // IPair implementation accessible within view
   }
}
```

Figure 7.12: A view implemented with a Java inner class.

In the SIR model, views are introduced at the type level, not at the class level. In Java an operation to create a new view must be introduced at the interface level with inner classes being introduced in variants to implement the view creation operation.

7.6.2 Extending Java for View Support

The problem associated with implementing views in Java is to support state on a per–view instance while preserving identity in some way. Additionally, extension methods (for example, before and after methods) with full delegation semantics should be supported.

This can be done by using the inner class technique described above as an implementation pattern and using meta–object protocol techniques to transform a program written in SIR–specific notation into one using the implementation pattern. The user would then be able to write programs such as the one in Figure 7.13.

```
interface Pair
{

}

interface Point
{

}


class IPoint variantOf Pair
             viewOf    Point
{

}
```

Figure 7.13: SIR notation for views.

This approach supports both per–view states, preservation of identity and access to target methods from the view. It is also possible to distinguish between a method on the view and a method with

the same name on the target.

In the implementation of a view it must also be possible to add before and after methods to extend target methods that are accessed via the view. Handling method extension for view operations can be handled in the same way as for the variant relationship, using meta–object protocol techniques. There is one additional case that must be handled for the view relationship: when an extended method is invoked from a method in the target, but via a view, then the extended method must be invoked. Since the view is not a subclass of the target we do not have overriding semantics and method extension must be simulated via meta–object protocol techniques.

7.7 Support for Evolution

Evolution allows the definition of an abstraction to be statically replaced without modification to the original definition. We consider the current, limited, support for evolution in Java and then consider a language extension that supports evolution.

7.7.1 Current Support for Evolution in Java

Evolution cannot easily be achieved within the Java language. One approach would be to rename an existing class and then introduce a subclass with the original name of the superclass, containing details of an adaptation, as a subclass. If, for example, we have a Point class, representing 2–D points, and a new requirement is introduced for Point to support printing to file, then we could rename Point as PointOriginal, and create a new Point subclass of PointOriginal with the additional behaviour. This approach requires additional work if the class to be evolved has class methods which are not inherited like other methods. Renaming the original version of a class requires access to the source code and modification of the original abstraction so it cannot be said to support evolution. However, this approach does support conformant modification of an existing abstraction with the separation of new elements from existing elements. This approach would enable solutions designed within the SIR model to be implemented.

7.7.2 Extending Java for Evolution Support

Evolution is possible using the *System–Mixin* plug–in of the Extensible Pre–Processor (EPP) system [Yuuji Ichisugi, 1999a] which is a tool for meta–level programming in Java. The notion of a system–mixin is similar to that of evolution. A base class is defined and other 'system–mixins' which augment the functionality of that class can be implemented. It is also possible to specify dependencies between adaptations (system–mixins) so that required adaptations will be loaded before the adaptation that requires them. Figure 7.14 provides an example of the system–mixin syntax used by EPP (taken from [Yuuji Ichisugi, 1999b]).

A pre–processor could be used to convert SIR–specific syntax to that used by EPP. In the SIR model there should be no need to use any special syntax to create an interface or type that can be

```
SystemMixin Skeleton {
  class Foo {
    define void m(String d){
      doDefault();
    }
  }
}

SystemMixin A {
  class Foo {
    void m(String d){
      if (d.equals("A")) { doA(); } else { original(d); }
    }
  }
}

SystemMixin B {
  class Foo {
    void m(String d){
      if (d.equals("B")) { doB(); } else { original(d); }
    }
  }
}
```

Figure 7.14: EPP System–Mixin Example.

evolved since all types and classes can be evolved. Additionally the appropriate syntax for defining an adaptation of an existing type/class would use an `adapts` keyword, as shown in Figure 7.15.

```
type Point
{
   move(int x, int y);
}

type PointPrint adapts Point
{
   print();
}
```

Figure 7.15: Evolution syntax.

Note that EPP system–mixins also provide support for the state–based abstractions technique discussed in Chapter 5. A new adaptation can handle an operation for certain states and make a call to super in other cases so that other cases can be handled, this is shown in the `d` method in Figure 7.14.

7.8 CASE Tool Support

Direct support for SIR constructs in a programming language may seem like the most effective way to achieve a seamless software development lifecycle. However, the current trend in software development is towards CASE tool support for automated code generation. All of the major OO CASE tools support automated code generation to some degree. With this in mind it becomes more important that the SIR constructs are supported at the CASE tool level and less important that direct language support is provided.

If CASE tools are generating much of the code for a system and additional code is written within

the CASE tool environment then what is necessary is a set of patterns for implementing the SIR constructs in the programming language being generated. When code generation is automated, a greater degree of complexity in the patterns that map from the design-level constructs to the programming language is acceptable. Some of the techniques illustrated in the previous sections which are too complex for hand-coding would be useful in such an environment.

In such a scenario it is important that any future modification, extension or reuse, of a system is done within a CASE tool environment where the design of the system is available. Currently, when a class library is available for reuse its the design is not usually made available or it is only available as a document — the design cannot be loaded into a CASE tool. The fact that different case tools use different representations for models means that models are not interchangeable between CASE tools even if they were to be distributed with libraries.

Recent developments in CASE tool technology should improve this situation. The UML language is becoming the *de facto* standard for describing software systems. Current work is looking at encoding UML models in XML, a text–based format for the sharing of structured information. Having an encoding for models, such as XMI [Unisys Corporation and partners, 1998], will make it feasible to transport designs along with systems. Since the SIR model is expressed in terms of the UML metamodel it would also be possible to capture the semantics of an SIR system in an XML–encoding.

This would make it possible to load the design of a binary–only component into a CASE tool and design a system that depends on it. This is a very valuable technique since it allows design information, such as that conveyed by the use of the SIR constructs, to be available whenever software is used, reused or extended.

CASE tool support is a promising alternative to direct language–level support for the seamless integration of the SIR model into the software development lifecycle.

7.9 Conclusion

The techniques described in this chapter illustrate that it is feasible to implement the five SIR relationships in a statically–typed programming language. Techniques for adding SIR support to Java have been described, making use of existing extensions to Java and Java meta–object protocols. A possible future direction would be to build on the language constructs and techniques described in this chapter to design and implement a programming language with full SIR support.

An alternative to direct support at the programming level has also been outlined, support for the constructs could be added at the CASE tool level complete with automated code generation. In this scenario the code is not expected to be manipulated directly, instead it is always manipulated via a CASE tool. Developments in XML–encoding of UML models is expected to support such a scenario. It will be possible to transport designs along with source code or binary modules so that any development involving existing components can be carried out in the context of their design rather than their source code or public interfaces.

The two approaches could also be used in combination. Language support alone is not sufficient, CASE tool support during the design of systems would also be needed. Also, code generation from an SIR–aware case tool would be much more straightforward and less error prone if the translation was to a programming language with direct support for the SIR constructs.

Chapter 8

Conclusion

To date, the use of inheritance relationships in object–oriented software development has failed to provide all the benefits that it was supposed to offer — simply using inheritance does not lead to understandable, maintainable systems with a high level of reuse. Many years' practical experience of inheritance has shown us that it is frequently misunderstood and difficult to master. Inheritance can offer considerable benefits, but only when it is used well.

A further problem is that there is no single accepted form of inheritance, there are multiple applications of inheritance that are considered to be valid modelling techniques. The overloaded nature of the current concept of inheritance means that even when inheritance has been used effectively — to model an underlying conceptual relationship — that relationship is lost in the translation to a single design–level relationship. The flexibility of inheritance makes it a powerful modelling tool *and* a powerful means of producing opaque, hard–to–maintain systems.

The goal of this thesis has been to address the highly problematic nature of inheritance, by first providing a comprehensive understanding of object–oriented inheritance, then using this understanding to build a structured framework in which inheritance can be applied effectively.

8.1 Structured Inheritance Relationships

Previous classifications of inheritance, such as [Budd, 1997] and [Meyer, 1996], have highlighted a number of distinct uses for inheritance mechanisms. At present, even if a developer has a clear understanding of the conceptual relationship underlying a particular application of inheritance, that understanding does not make it into the software design — this means that this information is not available for future maintenance, extension or reuse of the system. Existing classifications of inheritance go some way towards improving this situation since they allow particular uses of inheritance to be labelled with meaningful names which correspond to informal definitions. However, these classifications simply attempt to explain *existing* uses of inheritance mechanisms in current programming languages, and they do not claim to be exhaustive even within that context.

In developing a fundamental model of inheritance, it is not sufficient to examine only how current inheritance mechanisms are being used. It is essential to focus on the underlying conceptual relationships that it is desirable, and appropriate, to model at the design stage, using inheritance relationships. This goal–directed approach removes the restrictions of current inheritance mechanisms, and allows us to take conceptual relationships to their natural extensions.

In Chapter 3, we developed the fundamental conceptual relationships that underly appropriate uses of inheritance, identifying five key structured inheritance relationships (SIRs). These five SIRs encapsulate the relationships that can occur between a subclass and a superclass. Each of these structured inheritance relationships has been developed with a sound conceptual basis which enables the possible inheritance relationships between abstractions to be precisely specified.

Specialisation replaces the confused notion of "is–a" which is currently, and inadequately, used as a guide for the appropriate use of inheritance. Here we have defined specialisation as allowing further detail to be added to an existing supertype definition to create a more specific subtype with fewer potential instances. For example, we might specialise a Customer abstraction in a billing application to create a CorporateCustomer subtype with all the features of a general customer and specific details associated with corporate customers. Specialisation operates between types (which cannot contain implementation details) and leads to syntactic and behavioural substitutability. The supertype in a specialisation relationship must be a genuine part of the definition of the subtype, for example, being a customer is clearly an intrinsic part of being a corporate customer. This may appear to be an overly restrictive interpretation of the "is–a" relationship, but the SIR model provides other relationships with other semantics so there is no need for a more general is–a relationship, encompassing all valid uses of inheritance.

The variant relationship allows realization details to be added to an incomplete superclass definition. The subclass in a variant relationship may provide a complete realization of an abstraction, or it may provide partial realization to be completed by its own subvariants. The variant relationship allows alternate realizations of types to be built up layer by layer with each variant relationship representing an implementation decision. In this way, variant supports behavioural variation across abstractions that are classified in the same way. It is not difficult to recognise that two objects presenting the same interface to the world (and therefore being indistinguishable by clients) need not necessarily have the same internal behaviours. For example, we might have an ActiveDocument abstraction which knows how to display itself — all documents have a display method, but variants for Word documents, Postscript files, HTML files and PDF files will all have different internal behaviours. Since variants do not introduce new external behaviours, clients can access the full functionality of each of the variants through the same interface and new variants can be introduced without modification to clients.

The view relationship is based on a particular use of multiple inheritance, which attempts to offer different interfaces to the same object, so that clients can choose to use the one that best meets their requirements. Specifically, the view relationship enables the development of a subclass which provides an alternative set of behaviours to be offered to certain clients. For example, we might

develop a Taxi abstraction as view of a Car abstraction, so that a car that is currently a taxi can be viewed as a taxi by its clients. Conceptually, views often correspond to the different rôles that an abstraction may play throughout its lifetime. Providing a specific view relationship allows us to model such situations precisely. The view relationship is a powerful technique that enables an abstraction to be reused, by keeping individual abstractions simple and manageable, but allowing them to be extended to suit new kinds of client.

The construction relationship allows existing implementations to be reused in the realization of a new abstraction. For example, we might construct a Window abstraction from a MenuBar abstraction with the Window receiving events when menu items are selected. This is a very common use of inheritance, but it has often been advised against. This is primarily because it does not correspond to an is–a relationship and using the same mechanism — inheritance — to model two completely different conceptual relationships leads to 'spaghetti inheritance' hierarchies which are difficult to understand and maintain. Inheritance is widely used for construction–like relationships so there is clearly a need for such a relationship and disallowing it would be inappropriate. It is preferable to clearly specify the relationship and support it as a valid form of inheritance. Construction provides code–level reuse based on conceptual sharing of behaviour — when abstractions related by construction appear to have similar behaviours this is because they really do have the same realization. Importantly, the reuse of behaviour via construction does not introduce a spurious and conceptually incorrect subtyping relationship between the reusing class and the class that provides the realization. The SIR model clearly separates the various forms of inheritance so that the spaghetti inheritance problem does not arise.

Evolution has been developed from the understanding that most software systems are a conglomeration of efforts by many people over a period of time. Since systems are practically guaranteed to change over time, it is necessary to provide a means by which abstractions may evolve as their environment changes. For example, a BankAccount abstraction may need to be evolved when the bank changes its policies. Additionally, abstractions developed for one system may require adaptation when they are reused within the context of another system. Evolution allows the implementation of a complex abstraction to be built up in stages, clearly separating the behaviour that is introduced at each stage. Through evolution, an existing abstraction can be extended, this is a distinct improvement on the current approach of creating an improved subclass to meet new requirements but also keeping the existing inadequate abstraction. By using evolution we ensure that only the required abstraction appears in the new system and we still benefit from reusing the existing superclass definition.

8.2 A New Model of Inheritance

Developing a conceptual basis for the relationships that can replace the current confused notion of inheritance is simply the first stage in providing a new model of inheritance. In Chapter 4 we built on the informal definitions of Chapter 3 to provide a detailed semantics of each SIR. The resulting

SIR model of inheritance uses the SIRs as fundamental building blocks which can be combined to model more complex forms of inheritance.

Considering the SIRs as a basis for all applications of inheritance means that the relationships are considered as a whole, rather than each one being developed in isolation. The resulting model of inheritance provides a complete design–level replacement for the current overloaded and confused notion of inheritance.

The resulting SIR model of inheritance supports the development of structured inheritance frameworks, rather than the ad–hoc 'spaghetti inheritance' hierarchies that result from the undisciplined use of inheritance.

8.3 Techniques for Disciplined Software Construction

The SIR model provides an ideal starting point for describing the appropriate application of inheritance within software construction. Like design patterns, the SIR techniques capture best practice for particular scenarios. The five specialised SIRs support the description of modelling techniques at a high level. Without the SIR relationships, each use of inheritance within a technique or pattern must be disambiguated through detailed explanation. However, since the SIR relationships each have precise semantics they can be used to provide succinct descriptions of inheritance–based techniques.

Chapter 5 provided a set of techniques that represent best practice in the SIR model. These techniques demonstrate how the SIR model can be used to develop software architectures that exhibit extensibility, modifiability, reusability and reuse — benefits that inheritance was originally intended to confer. The techniques illustrate that the SIR model of inheritance has sufficient expressive power to describe solutions to traditional modelling problems, and that the higher–level starting point leads to manageable solutions to relatively complex problems. At the same time, the clear conceptual basis for the SIRs means that the resulting solutions are readily understandable, with each use of inheritance playing a well–defined rôle in the overall system design.

Additionally, because each of the SIRs is a clearly defined conceptual relationship, it is possible for the developer to gain an understanding of scenarios which, in the past, have been difficult to model using inheritance. For example, the relationship between points and coloured points, and between squares and rectangles, both of which are frequently used to illustrate the difficulties associated with used inheritance. In Chapter 6, a detailed analysis of both of these well–known modelling problems was presented, showing that the SIR model can provide a suitable basis for solving inheritance problems in software development.

8.4 Inheritance for Reuse

It should be noted here that, as mentioned earlier, one of the problems with the current muddled view of inheritance is that it has failed to generate the high degree of reuse that was expected. There

are a number of ways in which the SIR model improves upon this situation.

One of the problems with using inheritance for reuse has been the tight–coupling that occurs between superclass and subclass. This leads to complications such as the fragile base class problem discussed in Chapter 2. The SIR model improves this situation since it provides multiple forms of inheritance that are suitable for different circumstances. For example, it is possible to create a new variant of an existing type without inheriting any associated implementation, this means that it is not necessary for a subclass to depend on superclass implementation in order to be used by supertype clients. Additionally, construction provides a way of reusing implementation, with inheritance–related benefits, but without the problems associated with tight–coupling.

The likelihood of reuse, via inheritance or otherwise, is reduced if an existing abstraction is overly complex — it may seem simpler to build a new abstraction from scratch. The view relationship is valuable in this context. Firstly, it supports the development of reusable abstractions by placing only core functionality into an abstraction and thereby making that abstraction more understandable and reusable, additional functionality which may not be reusable in the same context can be placed in separate views. Secondly, view allows an abstraction to be reused in the context of new clients by introducing behaviour specific to those clients. This means that abstractions easily be reused in context–specific scenarios.

A further problem with using inheritance for reuse is that an existing abstraction may not be general enough to meet the requirements of a new situation. The developer of the original abstraction may have unintentionally placed unnecessary restrictions on the abstraction which prevent it from being reused in a particular scenario. Evolution allows an existing abstraction to be made more general so that it can meet the requirements of new clients whilst still maintaining its contract with existing clients. This means that we have a mechanism for reusing. and improving upon, existing abstractions, even when the original version is overly restrictive for a new application.

In addition, since each application of inheritance within an SIR system has a clear meaning, the understandability of such systems is improved, and this in itself simplifies the process of reuse.

8.5 Understanding of Inheritance

The development of the SIR model has provided a number of insights into the nature of inheritance which further current understanding.

The SIR model gains some of its power from combining the advantages of class and object level inheritance. All specification of inheritance relationships is at the class level but the construction and view relationships are instantiated at the object level. This is in contrast to previous approaches in which either class or object level inheritance is selected. The SIR combined approach offers structured modelling by introducing all inheritance relationships at the abstraction (class) level, while flexibility is achieved by allowing superclass instances to be assigned dynamically and shared by multiple subclasses.

The SIR model of inheritance — with five fundamental inheritance relationships rather than a single overloaded notion of inheritance — permits a more detailed analysis of when multiple inheritance is appropriate. In Chapter 4, both homogeneous and heterogeneous combinations of inheritance were considered. The resulting detailed classification illustrates how complex forms of inheritance can be built from the fundamental SIRs, and provides a comprehensive conceptual basis for the use of multiple inheritance. A key insight is that multiple inheritance of implementation with overriding, which is represented by the variant relationship and introduces strong coupling between abstractions, should be restricted to cases where the superclasses have a common ancestor type. Where superclasses are not related in this way, multiple variant is not possible, and construction which provides weaker coupling (both conceptual and semantic) is often the appropriate relationship.

Similarly, a more detailed analysis of the Abstract Superclass Rule (ASR) was made possible by the SIR model of inheritance. The meaning of the ASR within the SIR model was considered, leading to techniques for achieving its advantages while avoiding the forward planning normally associated with the ASR — making use of the evolution SIR as a key software design relationship was important here.

Unlike most treatments of inheritance at the design level, the SIR model also considered recent developments in type theory including multiple dispatch and matching. These relationships have been introduced at the implementation level to overcome a lack of expressivity in current programming languages. Since design is typically constrained by the target programming language, it has not been possible to design systems with matching and multiple–dispatch characteristics. Matching highlights the need to distinguish between abstractions representing objects and abstractions representing type–related sets of objects. Multiple–dispatch supports the modelling of behaviour that is dependent on more that one object. By incorporating these concepts at the design level, the SIR model can be used to design systems for modern languages such as Kiev, Pizza and PolyTOIL.

8.6 Directions for New Research

There are a number of issues arising from the SIR model of inheritance that provide directions for new research. There are two main categories of new research resulting from the SIR model: firstly, the further development of the SIR framework to include programming language design and implementation, CASE tool support and mathematical formalism of the SIR model; and secondly, the development of SIR–based solutions to problems such as comparing the expressiveness of programming languages, assessing proposed inheritance–like constructs, and teaching of the object–oriented paradigm. We provide motivation for new research in each of these categories, including recommendations on how such research should proceed.

8.6.1 Measuring Programming Language Coverage

The SIR model of inheritance is more powerful than the inheritance mechanisms found in current programming languages. Since the SIR model provides a precise classification of inheritance rela-

tionships is it possible to use it as a basis for evaluating existing and newly developed programming languages.

The SIR relationships provide a set of criteria against which object–oriented languages can be measured. An object–oriented language that does not support all of the SIR relationships has less modelling power than one that does. The SIR model can thus be used as a basis for measuring the level of support for inheritance in object–oriented programming languages (or modelling languages). In this way, it is also possible to develop meaningful comparisons of programming languages.

8.6.2 Categorisation of New Language Constructs

Due to the problems with inheritance discussed in Chapter 2 there has been a great deal of research into inheritance–like constructs. That is, constructs that are intended to either replace or complement current inheritance mechanisms.

The SIR model could be used to develop a high–level taxonomy of such constructs. A new construct can readily be categorised as providing support for one or more of the SIRs. The SIR model can also be used to provide justification for the introduction of a new construct to a programming language that did not previously provide direct support for a particular SIR.

8.6.3 Teaching of the Object–Oriented Paradigm

The SIR framework could also be employed in the teaching of the object–oriented paradigm. As discussed in Chapter 2 inheritance is a difficult technique to master. Replacing a single complex mechanism with five simpler relationships with clear conceptual foundations allows a gradual learning path since the relationships can be introduced one at a time.

A thorough understanding of the SIR relationships is a much stronger foundation for object–oriented design than a general idea that inheritance corresponds to the notion of 'is–a' or a detailed understanding of the complex inheritance mechanism of C++.

8.6.4 SIR Programming Language

The five SIRs can largely be implemented in existing programming languages (as shown in Chapter 7 for the Java language) but this is not ideal. It would be preferable to move from a design using SIR constructs to an implementation that follows the same structure.

The development of a programming language that directly supports the SIR relationships by providing specialised constructs is advantageous for a number of reasons:

1. A programming language that directly supports the SIR relationships will be able to provide appropriate errors and warnings based on the semantics of the SIR relationships (for example preventing primary types from referring to secondary types).

2. Direct support for constructs allows them to be implemented in an efficient manner which may not be possible in a language without such support.

3. Implementation is more straightforward and less error prone if the structure of the design can be mapped directly into the implementation.

4. Modifications to the design result in similar modifications to the code when the structure is the same.

5. Understanding the design goes a long way towards understanding the code and vice–versa.

Chapter 7 of this thesis provides much of the material required to develop an extension of the Java language supporting the SIR model.

8.6.5 CASE Tool Development

Although programming language support for the SIR relationships is important it would be even more beneficial to have support at the design stage, when constructing a system model.

This is especially true if much of the code for the resulting system is automatically generated — the difficulty of mapping from design–level constructs to programming–level constructs can be encapsulated into the CASE tool rather than managed by hand. In such an environment any modification to the system will be carried out within the CASE tool environment where the design is present. In this situation direct support for the SIR model within the case tool is beneficial. Tasks performed by a case tool would include:

1. Providing a current representation of an abstraction based on the adaptations that apply to it in the system under consideration.

2. Generation of complete descriptions of abstractions based on the various possible views of an abstraction.

3. Presentation of a selection of construction units that implement a particular type and may be of use in the implementation of a new abstraction that wishes to offer the functionality associated with that type.

4. Preventing the creation of invalid inheritance relationships (such as specialisation relationships that introduce implementation and construction relationships that override inherited methods).

5. Hiding certain parts of the model, for example hiding variants to give a high–level view of the system.

The Argo/UML CASE tool [Robbins and Redmiles, 1999] which has been developed in a research environment and is available under an open source license, would make a good basis for the investigation of CASE tool support for the SIR model. Argo/UML provides support for software design using the UML and automatic code generation; the tool could be used as a vehicle to investigate support for the SIR model at the CASE tool and code generation levels.

8.6.6 Mathematical Formalism

The SIR framework would benefit from a mathematical underpinning. Possible vehicles for this work would be an order–sorted algebra (see [Meseguer and Goguen, 1993]) or category theory which supports the modelling of higher–order abstractions (see [Piessens and Steegmans, 1996]). The development of a mathematical formalism for the SIR model would provide a sound theoretical basis to support the strong conceptual basis developed in this thesis.

A further approach to formalising the SIR model would be to develop a formal specification language supporting the five SIR relationships. Such a language could be based on an existing object–oriented formal specification language such as Z++ [Lano and Haughton, 1994], VDM++ [Lano, 1995] or Larch [Guttag et al., 1993]. The development of an SIR formal specification language would enable the semantics of the SIR model to be precisely expressed.

Although such formalisms would now be valuable, it was important to develop the conceptual foundation for the SIR model first. This approach has prevented the adoption of mechanisms simply because they are 'theoretically tidy'. The conceptual model should guide the formal model and not the other way around.

8.7 A Final Word

The SIR model of inheritance developed in this thesis shows that much can be done to improve the current confused notion of inheritance. The SIR model does not necessarily represent the only way in which a basis for inheritance can be developed but it does offer an improvement over the incomplete classifications that are currently available.

We have highlighted the power of inheritance as a software modelling tool — inheritance is too important a technique to simply avoid or only use in one or two restricted scenarios. Instead of discarding inheritance because of its associated problems, these problems have been directly addressed. This thesis has drawn together a large body of inheritance–related research in the areas of cognitive modelling, software design methods and techniques, and programming language theory and practice. This work has been consolidated and built upon to provide a new model of inheritance in which five structured relationships — specialisation, variant, view, construction and evolution — are recognised as fundamental building blocks. The five relationships have been shown to be necessary, conceptually orthogonal and sufficient to replace the current overloaded notion of inheritance.

The SIR model has advanced the understanding of inheritance as a conceptual modelling and design technique, and provided a framework in which the structured use of inheritance leads to well–designed systems in which the semantic intentions of developers are clearly expressed.

Appendix A

Glossary of Terms

General Object–Oriented Terminology

abstraction

A concept from the application domain or realization domain of a system.

abstract class

A class that cannot have direct instances.

abstract data type (ADT)

A type defined in terms of the operations that are available on it.

abstract type

A type which has no direct instances. All instances of an abstract type are instances of one of its (non trivial) subtypes.

assertion

A general term for *preconditions*, *postconditions* and *invariants*.

attribute

A variable that forms part of the definition of a class. Also known as a data element.

behavioural subtyping

A relationship between a subtype and a supertype in which a client expecting a supertype object will be satisfied if given a subtype object.

bounded genericity

Genericity is said to be bounded when the type parameter is constrained to be a subtype of a given type.

class

An abstraction with associated implementation.

class–based

A category of languages in which classes are used as patterns for sets of related instances.

client

An object which uses the services of another object which is referred to as the server.

clientship

The relationship between a client and a server in which the client makes use of the services offered by the server.

constructor

A special method for creating instances of a class.

contravariant

A type in a subclass interface is said to be contravariant if it varies in the opposite direction to the receiver, that is, it is more general than the corresponding type in the superclass.

covariant

A type in a subclass interface is said to be covariant if it varies in the same direction as the receiver, that is, it is more specific than the corresponding type in the superclass.

delegation

An instance–level relationship in which one object inherits the behaviours of another.

design–by–contract

A style of software specification in which the relationship between client and server is seen as a contract. Provided the client meets the a method precondition, then the server will meet the method postcondition.

domain

The set of concepts associated with a particular application area.

dynamic binding/late binding

A mechanism by which the appropriate method for a particular invocation is selected at run–time rather than compile–time. Dynamic binding is required to support inclusion polymorphism.

dynamic type

The dynamic type of an object is its most specific type, that is, the type of which it is a direct instance.

dynamically typed

Dynamically typed languages perform some or all type–checking at run–time meaning that run–time type errors are possible.

genericity

A mechanism for creating classes/types from parameterised classes/types by providing the missing type parameters.

inclusion polymorphism

The ability for a operation invocation to lead to a different method being invoked depending on the dynamic type of the receiver.

instance

An object created from a particular class is an instance of that class. It is also an instance of the type which the class implements.

implementation

The details of a class that are not present its external interface.

interface

The public operations on a type or class.

invariant

A condition associated with a class that always holds for instances of that class.

is–a

A relationship between abstractions in which an instance of one abstraction is also an instance of the other abstraction.

Liskov substitution principle (LSP)

The notion that subtypes should be behaviourally, as well as syntactically, substitutable for supertypes.

matching

A mechanism which supports polymorphism across types that are parameterised by type of the receiver.

method

The implementation of an operation.

method extension

A method which is invoked as well as the method which it extends.

multimethod

(One of) a collection of methods with the same name and number of arguments which are selected between at run–time based on the dynamic types of the arguments.

multiple dispatch

A mechanism that allows the dynamic types of arguments, other than the receiver, to contribute to method selection at run–time.

multiple inheritance

Multiple inheritance occurs when a subclass is in an inheritance relationship with two or more superclasses.

mutable

An abstraction is mutable if its instances can hold different values at different times. The value of an instance of an immutable abstraction is fixed at creation time and cannot be modified. Object–oriented classes are usually mutable.

object

A uniquely identifiable entity combining data and behaviour. In object–oriented languages an object is an instance of a class.

object–based

A category of languages in which objects are defined directly rather than as instances of classes.

operation

A unit of behaviour associated with an abstraction. An operation has an associated name, argument types and return type.

overloading

Overloading occurs when different methods could be selected as a result of the same operation invocation depending on the static types of the arguments or the receiver.

overriding

Overriding occurs when a subclass replaces an inherited superclass method. Subclass instances will use the overridden version of the method rather than the inherited version, even when the method is invoked from another inherited superclass method.

precondition

A assertion that describes the required state of an object before an operation can be invoked.

polymorphic assignment

An assignment in which a variable is made to hold an instance of a subtype of its declared type.

polymorphism

Method selection based on the dynamic type of an object held in a supertype variable.

postcondition

A assertion that describes the resulting state of an object after an operation has been invoked.

realization

The implementation or representation within a class, or the process by which implementation or representation is added in a subclass.

receiver

The object upon which an operation is invoked is said to be the receiver of that invocation.

refinement

The process by which implementation or representation is added to a specification in order to make it executable.

repeated inheritance

Repeated inheritance occurs when a subclass inherits from the same superclass more that once.

self type

A term used within a type definition to refer to the type being defined.

specification

The specification of an operation is its full definition including preconditions and postconditions. The specification or a class or type is the set of all operation specifications plus the invariants of the type or class.

static type

The static type of an object is the declared type of the variable in which it is held (which must be a supertype of the dynamic type of the object, including the trivial supertype). The static type of an object is therefore dependent on context.

static typing

In statically typed languages, all type–errors are detected at compile–time so that run–time type errors cannot occur.

subclass

The class which is the recipient of inherited characteristics in an inheritance relationship.

superclass

The class which is the source of inherited characteristics in an inheritance relationship.

substitutability

The property that allows an abstraction of one kind to be used as if it were an abstraction of another kind.

subtype

The more general type in a subtyping relationship.

subtyping

The relationship that allows subtype instances to be, syntactically or behaviourally, substituted for supertype instances.

supertype

The more specific type in a subtyping relationship.

type

A description of a set of domain objects with the same characteristics.

unbounded genericity

Genericity is said to be unbounded when the type parameter is not constrained.

variable

A named reference to an object. The declared type of a variable determines the objects that can be assigned to it.

SIR Terminology

adaptation

The abstraction describing the differences between the existing abstraction and the resulting abstraction in an evolution relationship.

construction

An inheritance relationship which allows existing implementation to be reused in the realization of a new abstraction.

construction unit

The superclass in a construction relationship.

evolution

An inheritance relationship which supports the adaptation of an existing abstraction in order to meet changing requirements.

inheritance (relationship)

A stated, continuing, relationship between two abstractions in which the characteristics of the first abstraction, the superclass, are also characteristics of the second abstraction, the subclass. Also known as SIR or SIR inheritance.

primary type

A type representing an abstraction in the application domain of a system, rather than in the realization domain.

secondary type

A type representing an abstraction in the realization domain of a system. A secondary type is automatically created when an implementation class is defined.

specialisation

An inheritance relationship which allows objects to be viewed at different levels with higher levels providing less detail and therefore encompassing more objects.

target

A superclass in a view relationship.

variant

An inheritance relationship which supports the modelling of differences between sets of objects that do not cause them to be classified differently.

variant

A subclass in a variant relationship.

view

An inheritance relationship which supports client–specific behavioural variation for abstractions.

view

A subclass in a view relationship.

References

America, P. [1989]. A portable implementation of the programming language POOL, *in* J. Bézevin (ed.), *TOOLS EUROPE 1989*, pp. 347–353.

ANSI [1983]. Military standard: Ada programming language, *Technical report*, ANSI and US Government Department of Defence, Ada Joint Program Office. ANSI/MIL-STD-1815A-1983.

Armstrong, J. M. and Mitchell, R. J. [1994]. Uses and abuses of inheritance, *Software Engineering Journal* **9**(1): 19–26.

Arnold, K. and Gosling, J. [1996]. *The Java Language Specification — Fourth Printing*, Addison-Wesley.

Bennett, D. [1997]. *Designing Hard Software: The Essential Tasks*, Manning, Prentice Hall.

Biddle, R. and Tempero, E. [1995]. Understanding OOP language support for reusability, *Workshop on Institutionalizing Software Reuse (WISR 7)*.

Birtwhistle, G. M., Dahl, O.-J., Myhrhaug, B. and Nygaard, K. [1973]. *Simula Begin*, Petrochello/Charter.

Black, A. and Palsberg, J. [1994]. Foundations of object–oriented languages, *ACM SIGPLAN Notices* **29**(3): 3–11.

Booch, G. (ed.) [1994]. *Object-Oriented Analysis and Design with Applications*, Benjamin Cummins.

Booch, G., Jacobson, I. and Rumbaugh, J. [1997]. *UML Semantics version 1.1*, Addison–Wesley. http://www.rational.com/uml/resources/documentation/semantics/.

Brachman, R. J. [1983]. What is-a is and isn't: an analysis of taxonomic links in semantic networks, *Computer* **16**(10): 30–36.

Bruce, K., Cardelli, L., Castagna, G., The Hopkins Object Group, Leavens, G. T. and Pierce, B. [1995]. On binary methods, *Theory and Practice of Object Systems* **1**: 221–242.

Bruce, K., Schuett, A. and van Gent, R. [1995]. PolyTOIL: A type–safe polymorphic object–oriented language, *European Conference on Object–Oriented Programming*, Vol. 952 of *LNCS*, pp. 27–51.

Budd, T. [1997]. *An Introduction to Object-Oriented Programming*, second edn, Addison Wesley.

Bunce, T. and Stein, L. [1998]. CGI:: modules for Perl 5, Web pages/Perl scripts. Available as: http://stein.cshl.org/WWW/software/CGI::modules/.

Castagna, G. [1995]. Covariance and contravariance: conflict without a cause, *ACM Transactions on Programming Languages and Systems* **17**(3): 431–447.

Chambers, C. [1997]. The Cecil language — specification and rationale, *Technical report*, Department of Computer Science and Engineering, University of Washington.

Chambers, C., Ungar, D., Bay-Wei, C. and Hölze, U. [1991]. Parents are shared parts of objects: inheritance and encapsulation in Self, *Lisp and Symbolic Computation* **4**(3).

Ciancarini, P. and Cimato, S. [1997]. Specifying component–based software architectures, *in* L. G. T and M. Sitaraman (eds), *Foundations of component–based systems workshop.*

Cox, B. [1986]. *Object–Oriented Programming: An Evolutionary Approach*, Addison–Wesley.

Edwards, S. H. [1993]. Inheritance: One mechanism, many conflicting uses, *Sixth Workshop on Institutionalizing Software Reuse (WISR 7).*

Firesmith, D. [1995]. Inheritance guidelines, *Journal of Object–Oriented Programming* **8**(2): 67–72.

Fowler, M. and Scott, K. [1997]. *UML Distilled — Applying the standard object modeling language*, Addison Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. [1993]. Design patterns: Abstractions and reuse of object–oriented design, *ECOOP '93 Conference Proceedings.*

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. [1995]. *Design Patterns: Elements of Reusable Object–Oriented Software*, Addison Wesley.

Gil, J. and Lorenz, D. H. [1996]. Environmental acquisition — a new inheritance–like abstraction mechanism, *OOPSLA '96.*

Goldberg, A. and Robson, D. [1983]. *Smalltalk-80: The language and its implementation*, Addison-Wesley.

Guttag, J. V. et al. [1993]. *Larch: Languages and Tools for Formal Specification*, Springer–Verlag.

Halbert, D. C. and O'Brien, P. D. [1987]. Using types and inheritance in object–oriented programming, *IEEE Software* **4**(5): 71–79.

Hürsh, W. L. [1994]. Should superclasses be abstract?, *ECOOP '94 Proceedings*, pp. 12–31.

Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G. [1994]. *Object-Oriented Software Engineering*, Addison-Wesley.

Kizub, M. [1998]. Kiev language specification, Web pages. Available as: http://www.forestro.com/kiev/kiev.html.

Kramer, R. [1998]. iContract: The Java design–by–contract tool, *Tools 98*.

Kuo, Y. S. [1995]. When to inherit and when not to, *International Journal of Software Engineering and Knowledge* **5**(3): 391–405.

LaLonde, W. and Pugh, J. [1991]. Subclassing $\neq$ subtyping $\neq$ is–a, *JOOP* **3**(5): 57–62.

Lano, K. [1995]. *Formal Object–Oriented Development*, Springer.

Lano, K. and Haughton, H. [1994]. *Object–Oriented Specification Case Studies*, Prentice Hall.

Lieberherr, K. [1996]. *Adaptive Object–Oriented Software: The Demeter Method*, PWS.

Liskov, B. [1974]. Programming with abstract data types, *ACM Conference on Very High Level Languages*, Vol. 9 of *SIGPLAN Notices*, pp. 50–59.

Liskov, B. [1987]. Data abstraction and hierarchy, *OOPSLA '87, Addendum to the Proceedings*, pp. 17—34.

Liskov, B., Curtis, D., Day, M., Ghemawat, S., Gruber, R., Johnson, P. and Myers, A. C. [1995]. Theta reference manual.

Liskov, B. and Wing, J. [1994]. A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems* **16**(6): 1811–1841.

Liskov, B. and Zilles, S. [1975]. Specification techniques for data abstractions, *IEEE Transactions on software Engineering* **1**(1).

Madsen, O. L., Møller-Penderson, B. and Nygaard, K. [1994]. *Object–oriented programming in the Beta programming language*, Addison–Wesley.

Magnusson, B. [1991]. Code reuse considered harmful, *Journal of Object–Oriented Programming* **4**(7): 8–8.

McGregor, J. D. and Korson, T. [1993]. Supporting dimensions of classification in object–oriented design, *Journal of Object–Oriented Programming* **5**(9): 25–30.

Meseguer, J. and Goguen, J. A. [1993]. Order-sorted algebra solves the constructor-selector, multiple representation, and coercion problems, *Information and Computation* **103**(1).

Meyer, B. [1986]. Genericity versus inheritance, *ACM SIGPLAN Notices* **21**(11): 391–405.

Meyer, B. [1987]. Reusability: The case for object–oriented design, *IEEE Software* **4**(2): 50–64.

Meyer, B. [1988]. *Object–Oriented Software Construction*, Prentice Hall.

Meyer, B. [1992]. Applying design by contract, *IEEE Computer* **25**(10).

Meyer, B. [1996]. The many faces of inheritance: A taxonomy of taxonomy, *IEEE Computer* **29**(5): 105–108.

Meyer, B. [1997]. *Object–Oriented Software Construction*, second edn, Prentice Hall.

Mezini, M. [1997]. Dynamic object evolution without name collisions, *in* M. Akşit and S. Matsuoka (eds), *ECOOP '93 Conference Proceedings*, Vol. 1241 of *LNCS*, Springer, pp. 190–219.

Minsky, M. [1981]. A framework for representing knowledge, *in* J. Haugeland (ed.), *Mind Design*, The MIT Press.

Morgan, C. [1994]. *Programming from Specifications*, second edn, Prentice Hall.

Mossenbock, H. [1993]. *Object–oriented programming in Oberon–2*, Springer–Verlag.

Odersky, M. and Wadler, P. [1997]. Pizza into Java: Translating theory into practice, *Symposium on Principles of Programming Languages*, ACM, pp. 146–159.

Oliva, A. [1999]. The design and implementation of Guaraná, *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*.

Paepcke, A. (ed.) [1993]. *The CLOS Perspective*, MIT Press.

Palsberg, J. and Schwartzbach, M. [1990]. Type substitutation for object–oriented programming, *ACM Conference on Object–Oriented Programming: Systems, Languages and Applications*, pp. 151–160.

Parnas, D. L. [1972]. A technique for software module specification with examples, *Communications of the ACM* **15**(5): 330–336.

Piessens, F. and Steegmans, E. [1996]. Categorical semantics for object–oriented data specifications, *in* S. J. Goldsack and S. J. H. Kent (eds), *Formal Methods and Object Technology*, Springer.

Porter, H. H. [1992]. Separating the subtype hierarchy from the inheritance of implementation, *Journal of Object–Oriented Programming* **4**(9): 20.

Powell, M. L. [1995]. Alternative perspectives on object–oriented technology, *Software Practice and Experience* **25**(S4): 131–141.

Quillian, M. R. [1967]. Word concepts: a theory and simulation of some basic semantic capabilities, *Behavioural Science* **12**: 410–430.

Robbins, J. E. and Redmiles, D. F. [1999]. Cognitive support, UML adherence, XMI interchange in Argo/UML, *The First International Symposium on Constructing Software Engineering Tools (CoSET '99)*.

Rumbaugh, J. [1993]. Disinherited! examples of misuse of inheritance, *Journal of Object–Oriented Programming* **5**(9): 22–24.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. [1991]. *Object-Oriented Modelling and Design*, Prentice Hall International.

Sakkinen, M. [1989]. Disciplined inheritance, *in* S. Cook (ed.), *Proc. ECOOP '89*, Cambridge University Press.

Seidewitz, E. [1996]. Controlling inheritance, *Journal of Object–Oriented Programming* **8**(8): 36–42.

Seiter, L. M., Palsberg, J. and Lieberherr, K. J. [1998]. Evolution of object behavior using context relations, *IEEE Transactions on Software Engineering* .

Shang, D. L. [1994]. Covariant specification, *ACM SIGPLAN Notices* **29**(12): 58–65.

Singh, G. B. [1994]. Single versus multiple inheritance in object oriented programming, *OOPS Messenger* **5**(1): 34–43.

Stroustrup, B. [1988]. What is object–oriented programming?, *IEEE Software* **5**(3): 10–20.

Stroustrup, B. [1991]. *The C++ Programming Language*, second edn, Addison–Wesley.

Sun Microsystems Inc. [1997]. Java inner classes specification, Web pages. Available as: http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html.

Szyperski, C. [1998]. *Component Software: Beyond Object–Oriented Programming*, Addison–Wesley.

Tatsubori, M. [1998]. OpenJava WWW page, Web pages. Available as: http://www.softlab.is.tsukuba.ac.jp/~mich/openjava/.

The Jamie Developers [1998]. Jamie - multiple subclassing for java, Web pages. Available as: http://www.list.org/jamie/.

UML Specification (draft), version 1.3 beta R7 [1999]. Web. Available as: http://www.rational.com/uml/resources/documentation/media/OMG-UML-1.3-Alpha5-PDF.zip.

Unisys Corporation and partners [1998]. XML metadata interchange (XMI), *Technical report*, Proposal to the OMG.

Yuuji Ichisugi [1999a]. EPP home page, Web pages. Available as: http://www.etl.go.jp/~epp/.

Yuuji Ichisugi [1999b]. (EPP) systemMixin plug–in user's manual, Web pages. Available as: http://www.etl.go.jp/~epp/edoc/edocalpha/SystemMixin.html.